O'REILLY®

TURING 图灵程序设计丛书

涵盖Android

RxJava反应式编程

Reactive Programming with RxJava

RxJava项目核心贡献者主持撰写，涵盖120多个具体样例，全面易懂的使用指南



[波兰] 托马什·努尔凯维茨 [美] 本·克里斯滕森 著

张卫滨 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

张卫滨

毕业于天津大学，软件开发工程师，拥有十余年相关经验。热爱马拉松和摇滚乐，喜欢探究和钻研新技术，译作包括《Spring实战》《Spring Data实战》《反应式Web应用开发》等。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING 图灵程序设计丛书

RxJava反应式编程

Reactive Programming with RxJava

[波兰] 托马什·努尔凯维茨 [美] 本·克里斯滕森 著
张卫滨 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

RxJava反应式编程 / （波兰）托马什·努尔凯维茨
(Tomasz Nurkiewicz)，（美）本·克里斯滕森
(Ben Christensen) 著 ; 张卫滨译. -- 北京 : 人民邮
电出版社, 2019.12
（图灵程序设计丛书）
ISBN 978-7-115-52400-3

Ⅰ. ①R… Ⅱ. ①托… ②本… ③张… Ⅲ. ①移动电
话机－应用程序－程序设计②JAVA语言－程序设计 Ⅳ.
①TN929.53

中国版本图书馆CIP数据核字(2019)第239623号

内 容 提 要

RxJava 广泛应用于 Android 应用程序的开发，得到了广大开发人员的青睐。其语法简洁，运行高效，未来有望成为主流的开发模式。本书主要内容包括：RxJava 的基本概念，RxJava 提供的诸多操作符，如何将 RxJava 用于自己的应用程序以及如何与它交互，如何将 RxJava 嵌入代码库的不同地方，如何从头到尾实现反应式应用程序，流控制，回压机制，基于 Rx 的应用程序的单元测试、维护以及问题排查等相关技术。本书还特别收录了 2.0 版本和 1.0 版本的异同比较。

本书适合中高级 Java 程序员和软件架构师阅读。

◆ 著　　[波兰] 托马什·努尔凯维茨　[美] 本·克里斯滕森
译　　张卫滨
责任编辑　张海艳
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：20.5
字数：485千字　　　　2019年 12 月第 1 版
印数：1－3 000册　　　　2019年 12 月北京第 1 次印刷

著作权合同登记号　图字：01-2019-6608号

定价：99.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

谨以此书献给 Paulina Ścieżka，她是我见过的最坦诚的人。

感谢她的信任和指导（不仅限于本书的写作）。

她给我的生活带来的改变超出了她的想象。

——托马什·努尔凯维茨

目录

本书赞誉

“这本书深入探讨了 RxJava 的理念和用法，以及反应式编程的通用知识。两位作者在实现和使用 RxJava 方面拥有丰富的经验。如果你想掌握反应式编程，那么没有比阅读这本书更好的方法了。”

——Erik Meijer，Applied Duality 公司总裁兼创始人

“对于现代 Android 应用程序来讲，高度状态化、并发和异步实现是基本的特性，而 RxJava 是管理它们的好工具。这本书既是一个渐进式的学习工具，也是一份随时可翻阅的参考资料，如果没有它，掌握 RxJava 这个库可能会非常困难。”

——Jake Wharton，Square 公司软件工程师

“托马什和本在使用简单的方式解释复杂的事物方面很有天赋，这就是这本书读起来很令人愉悦的原因。对于每一个想要掌握反应式编程和 RxJava 的 JVM 开发人员来说，这本书都是案头书。作者谈及了许多主题，如并发、函数式编程、设计模式和反应式编程。但是，这么多的内容并不会令读者却步，而是会引领读者循序渐进地掌握越来越高级的概念和技术。”

——Szymon Homa，高级软件工程师

译者序

随着 Spring 5 引入对反应式编程的支持，反应式编程模式在 Java 领域中也得到了前所未有的关注。其实在其他语言中，反应式编程的理念出现和应用得更早一些，比如 Angular 很早就将 RxJS 纳入 HTTP 请求等使用场景了。

Java 静态化的特点，导致一些函数式或回调式的编程模式很难便利地应用到 Java 程序员日常的开发中。正是在这样的背景下，RxJava、Reactor 等反应式编程框架得到了社区和广大开发人员的青睐。RxJava 广泛应用于 Android 应用程序的开发，借助 RxJava 开发的 Hystrix 更是成为了现代微服务应用程序的标准配置。反应式编程的特点在于语法简洁、运行高效、性能开销可控，未来将是一种主流的开发模式。

就目前来看，反应式编程在企业级应用中的直接应用还比较少。主要的困难有两个：一个是关系数据库的访问，另一个是网络请求的调用。这在很大程度上是因为 JDBC 和 HTTP 本身具有阻塞式的特点。不过情况正在改变：R2DBC 项目致力于将反应式编程 API 应用到关系型数据存储中，而 RSocket 项目则致力于提供一个符合反应式流语义的应用层协议。相信随着这些项目在技术和社区方面的不断成熟，反应式编程在开发人员的工具箱中会占据越来越重要的地位。

本书全面阐述了反应式编程的理念，以及这些理念是如何在 RxJava 中实现的。尤为难能可贵的是，除了从头构建完整的反应式应用程序之外，作者还细致阐述了如何向已有的应用程序逐步引入 RxJava 框架。

稍微美中不足的是本书撰写得较早，是基于 RxJava 1.*x* 版本的。虽然 1.*x* 版本非常成熟，而且得到了广泛的应用，但是现在的 2.*x* 版本是基于反应式流规范重新编写的。不过，从 RxJava 1.*x* 到 RxJava 2.*x*，核心的设计理念是一脉相承的。为了便于读者实现从 RxJava 1.*x* 到 RxJava 2.*x* 的迁移，经允许，我翻译了 RxJava 项目领导者 Dávid Karnok 的两篇关于 RxJava 2.0 的文章，并作为本书的附录 C，供读者参考。除此之外，我和《反应式设计模式》一书的译者何品分叉（fork）编写了本书的源代码，会逐步将本书中较为独立的章节的源码

升级为 RxJava 2.0。读者可以参见 https://github.com/ReactivePlatform/Reactive-Programming-With-RxJava，也欢迎各位读者一起参与这项工作。为了本书的完整性，书中的源代码不再进行单独调整。

在本书的翻译过程中我得到了何品的很多帮助和指导，在此表示感谢。同时感谢图灵公司的朱巍老师。另外，还要感谢我的家人，他们包容并且支持我把大量的业余时间都投入到了本书的翻译之中。

虽然在本书的翻译过程中我花费了较长的时间去斟酌和修改，但是限于我的知识水平，难免会有纰漏，欢迎读者指正。读者可以通过 levinzhang1981@126.com 或者微信 levinzhang1981 联系我。

希望这本书对读者有用，祝阅读愉快！

序

2005 年 10 月 28 日，微软新任命的首席架构师 Ray Ozzie 以邮件的形式给员工发布了一份著名的备忘录，即“互联网服务时代来临”（The Internet Services Disruption）。在这份备忘录中，Ozzie 概述了微软、谷歌、亚马逊和 Netflix 使用 Web 作为交付服务的主要渠道给世界带来的变化。

从开发人员的角度，Ozzie 为大型企业的管理者做了以下重要声明。

> 复杂性具有杀伤力。它会吞噬开发人员的生命力，让产品难以规划、构建和测试，带来安全性方面的挑战，而且还会给终端用户和管理员带来挫败感。

首先，我们要考虑的是 2005 年时大型 IT 企业非常喜欢复杂的技术，如 SOAP、WS-* 和 XML。当时“微服务”这个词还没有出现，没有简单的技术帮助小企业的开发人员管理异步组合复杂服务的复杂性，同时缺少处理失败、延迟、安全性和效率的技术。

对于我在微软的 Cloud Programmability 团队来说，Ozzie 的备忘录是一口能发出振聋发聩声音的警钟，提醒我们要专注于发明一种简单的编程模型，用于构建大规模异步和数据密集型的互联网服务架构。在经历过多次失败之后，我们团队终于明白，通过对同步集合的 Iterable/Iterator 接口进行二元化（dualizing）处理，可以得到一对表示异步事件流的接口，以及所有熟悉的序列操作符，比如 map、filter、scan、zip、groupBy 等，它们能够转换和组合异步的数据流，因此，2007 年夏天 Rx 应运而生。在实现过程中，我们意识到需要管理并发和时间，为此扩展了 Java executor 理念，提供了虚拟时间和协同重新调度的功能。

经过两年密集的编程马拉松，我们团队探索了大量的设计方案以供选择，并在 2009 年 11 月 18 日先推出了 Rx.NET。不久，我们将 Rx 移植到了针对 Windows Phone 7 的 Microsoft.Phone.Reactive 上，并且开始在其他语言（如 JavaScript 和 C++）中实现 Rx，还用 Ruby 和 Objective-C 提供了实验性版本。

在微软内部，第一个使用 Rx 的用户是 Jafar Husain，他在 2011 年加入 Netflix 的时候将这项技术带到了那里。Jafar 在 Netflix 大力宣传 Rx，最终为 Netflix UI 的客户端栈重建了全新的架构，以完全支持异步流处理。对于我们来说非常幸运的是，他将自己的热情“传

递”给了本·克里斯滕森。当时本正致力于 Netflix 中间层 API 的工作，因为 Netflix 在中间层使用 Java，所以本在 2012 年开始了 RxJava 的工作，并在 2013 年初将代码库转移到了 GitHub 上以便于持续的开源开发。微软中另外一个较早的 Rx 采用者是 Paul Betts，他在去 GitHub 工作之后成功地说服了 GitHub 的同事（如 Justin Spahr-Summers）实现并在 2012 年的春天发布了针对 Objective-C 的 ReactiveCocoa。

随着 Rx 在业界越来越流行，我们在 2012 年秋天说服微软 Open Tech 开源了 Rx.NET。此后不久，我离开微软并创建了 Applied Duality 公司，并将全部时间用于使 Rx 成为标准的跨语言和跨平台的 API，以异步实时处理数据流。

时间很快来到 2016 年，Rx 越来越受欢迎，使用范围也急速扩大。Netflix API 的所有流量都基于 RxJava，Hystrix 容错库也依赖于 RxJava，该库隔离了所有的内部服务流量。通过相关的反应式库 RxNetty 和 Mantis，Netflix 正在创建一个完全反应式的网络栈，用于跨机器和进程边界连接所有的内部服务。RxJava 在 Android 领域也取得了巨大的成功，SoundCloud、Square、NYT、Seatgeek 等公司的 Android 应用程序都在使用 RxJava，并为 RxAndroid 扩展库贡献了力量。NoSQL 厂商，如 Couchbase 和 Splunk，为它们的数据访问层提供了基于 Rx 的绑定功能。其他采用 RxJava 的 Java 库包括 Camel Rx、Square Retrofit 和 Vert.x。RxJS 被 JavaScript 社区广泛采用，并为 Angular 2 等流行框架提供了强大的支持。该社区维护了一个网站（http://reactivex.io/），在那里可以找到许多语言中 Rx 实现的相关信息，以及精美的弹珠图和 David Gross（@CallHimMoorlock）的阐述。

自诞生起，Rx 就随着开发者社区的需求和输入而不断发展。在 .NET 中，Rx 的最初实现只专注于转换异步事件流，并在需要回压的场景使用异步枚举（asynchronous enumerable）。由于 Java 没有对异步 await 的语言级支持，开发者社区使用反应式拉取的概念扩展了 Observer 和 Observable 类型，并引入了 Producer 接口。由于有许多开源贡献者，RxJava 的实现非常复杂，但是经过了高度优化。

尽管 RxJava 的细节与其他 Rx 实现有细微的差异，但是它依然是专门为全新的分布式实时数据处理领域的开发人员创建的，让他们能够专注于本质复杂性，而不会为偶发复杂性所折磨。本书深入探讨了 RxJava 的概念和使用，以及 Rx 的通用理念。两位作者在实现和使用 RxJava 方面拥有丰富的经验。如果你想学习 Java 反应式编程，没有比购买本书更好的方法了。

Erik Meijer

Applied Duality 公司总裁和创始人

前言

本书目标读者

本书适合中级和高级 Java 程序员。你应该非常熟悉 Java，但是并不需要预先掌握反应式编程的知识。本书中的很多概念都与函数式编程相关，不过你也无须预先了解这方面的知识。以下两类程序员都能从本书中获益。

- 想提升服务器性能或者想让移动设备的代码更具可维护性的软件工程师。如果你属于这一类的话，将会在本书中找到解决实际问题的理念和方案，还有切实中肯的建议。在这种情况下，RxJava 只是本书帮助你掌握的另一个工具。
- 好奇的开发人员。他们可能听说过反应式编程，尤其是 RxJava，并且想要真正地理解它。如果你属于这种情况，即使没计划在生产环境的代码中使用 RxJava，本书也会开拓你的视野。

另外，如果你是一名真正的软件架构师，本书很可能会对你有所帮助。RxJava 会影响整个系统的总体架构，所以值得去了解。即便你刚刚体验编程，也可以尝试翻阅本书的前几章，这几章介绍了基础知识。一些底层的理念是通用的，如转换和组合，它们并不是反应式编程特有的。

本·克里斯滕森的说明

2012 年，我正在为 Netflix API 实现一种新的架构。在这个过程中逐渐明确的是，为了实现目标，我们需要拥抱并发和异步网络请求。在探索实现方式的过程中，我遇到了 Jafar Husain，他努力向我“兜售”他在微软学到的名为“Rx”的方法。我当时虽然非常熟悉并发，但是依然按照命令式的方式进行思考，并且极度以 Java 为中心，因为 Java 一直是我的经济支柱，我在它上面花费了大量时间。

Jafar 向我“兜售”Rx 方法时，因其函数式编程风格，我很难理解这些理念，只能先搁置。

在数月的讨论之后，整体的系统架构不断成熟。在这个过程中，我和 Jafar 继续进行了多次白板会议，直到我掌握了这些理论原则，并且意识到 Reactive Extensions 的优雅和力量。

我们决定在 Netflix API 中采用 Rx 编程模型，最终创建了 Reactive Extensions 的 Java 实现，即 RxJava，它遵循了微软从 Rx.Net 和 RxJS 开始的命名约定。

在我从事 RxJava 开发工作的大约三年间，大部分工作都是在 GitHub 上以开放的方式完成的。我有幸与一个不断成长的社区一起工作，120 多位贡献者一起将 RxJava 变成了一个成熟的产品，它被用于很多生产系统中的服务器端和客户端。在 GitHub 上它获得了 15 000 多颗星，成为了排名前 200 名的项目之一，同时在使用 Java 的项目中排名第三。

Netflix 的 George Campbell、Aaron Tull 和 Matt Jacobs 在 RxJava 成熟的过程中起着关键作用，他们将 RxJava 从早期的构建版本变成了现在的样子，所做的工作包括添加了 `lift`、`Subscriber`、回压以及 JVM 多语言支持。Dávid Karnok 随后参与了进来，他提交代码的次数和行数都已经超过了我。他是项目取得成功的关键因素之一，并且已成为项目的领导者。

我必须要感谢 Erik Meijer，他在微软期间创建了 Rx。在他离开微软之后，我曾在 Netflix 就 RxJava 项目与他合作过。现在，我有幸和他一起在 Facebook 工作。和他在白板前花那么多的时间一起讨论是我的荣幸。Erik 这样的良师益友大大提高了我的思想水平。

在这个过程中，我还在很多会议上阐述了 RxJava 和反应式编程的相关内容，并结识了很多人，他们帮助我学习了关于代码和架构的更多知识。如果是自学的话，我不可能学到这么多。

Netflix 支持我在这个项目上花费时间和精力，并且在技术文档方面提供了支持。我自己可能永远也写不出这样的文档。如果没有"工作时间"的付出和拥有不同技能的许多人员的参与，这种成熟度和规模的开源项目是无法取得成功的。

在第 1 章中，我会试图讲解为何反应式编程是一种有用的编程方式，以及 RxJava 是如何具体实现这些原则的。

本书其余部分由托马什编写，他写得非常棒。我有机会审阅并提了一些建议，但这是他的书，从第 2 章开始他将负责详细内容的讲解。

托马什·努尔凯维茨的说明

2013 年，我在一家金融机构工作的时候，第一次接触 RxJava。当时，我们要实时处理大量市场数据。那时数据管道的组成是这样的：Kafka 传递消息，Akka 处理交易，Clojure 转换数据，还有一个自定义构建的语言在整个系统中传播变更。RxJava 是一个非常具有吸引力的选择，因为它有一个统一的 API，能够很好地处理不同来源的数据。

随着时间的推移，我尝试在更多的场景中使用反应式编程。在这些场景中，可扩展性和吞吐量都至关重要。按照反应式的方式来实现系统肯定要求更高，但是它带来的好处更为重要，包括更高的硬件利用率以及由此带来的能源节省。为了充分理解这种编程模型的优势，开发人员必须拥有相对易用的工具。我们认为 Reactive Extensions 在抽象、复杂性以及性能之间实现了平衡。

除非特别说明，本书讲述的是 RxJava 1.1.6 版本。尽管 RxJava 支持 Java 6 及更高版本，但是几乎所有的示例都用到了 Java 8 中的 lambda 语法。在讨论 Android 的第 8 章中，有些示例展现了如何在不支持 lambda 表达式的环境中处理烦琐的语法。话虽如此，本书并不会一直采用最简短的语法（比如方法引用），这主要是为了在适当的场景下提升代码可读性。

本书结构

如果你从头读到尾的话，将会收获最大。如果你没有那么多的时间，也可以选择最感兴趣的部分。如果某个概念在本书前面的章节中介绍过了，那么你很可能会找到关于参考章节的说明。如下是各章的简要介绍。

第 1 章简要介绍 RxJava 的起源、基本概念以及理念（本）。

第 2 章讨论如何将 RxJava 用于自己的应用程序，以及如何与它交互。这一章非常基础，但是理解一些概念是特别重要的，比如热源和冷源（托马什）。

第 3 章快速介绍 RxJava 提供的很多操作符，讲解一些具有表现力且强大的函数，它们是这个库的基础（托马什）。

第 4 章更加实用，展现如何将 RxJava 嵌入代码库的不同地方，还会简单介绍并发性（托马什）。

第 5 章内容更高级，阐述如何从头到尾实现反应式应用程序（托马什）。

第 6 章解释流控制中一个非常重要的问题，并介绍 RxJava 中的回压机制是如何解决该问题的（托马什）。

第 7 章介绍基于 Rx 的应用程序的单元测试、维护以及问题排查等相关技术（托马什）。

第 8 章展现一些 RxJava 应用程序，尤其是分布式系统中的程序（托马什）。

第 9 章重点介绍 RxJava 2.*x* 未来的计划（本）。

在线资源

本书所有的弹珠图均来源于 RxJava 的官方文档，它们基于 Apache 许可证 2.0 版本发布。

排版约定

本书使用了下列排版约定。

黑体字

　　表示新术语或重点强调的内容。

等宽字体（`constant width`）

　　表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

加粗等宽字体（**`constant width bold`**）

表示应该由用户输入的命令或其他文本。

等宽斜体（*`constant width italic`*）

表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是 http://bit.ly/reactive-prog-with-rxjava。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

来自本的致谢

如果没有托马什的话，这本书将不会存在，他编写了这本书大部分的内容。当然，还有我们的编辑 Nan Barber，他为我们提供了很多帮助并且非常有耐心，陪我们坚持到了最后。感谢托马什在 Twitter 上答复我寻找作者的消息，最终将这本书变成现实。

我还要感谢 Netflix 开源组织和 Daniel Jacobson 多年来对我本人和项目的支持。他们是这个项目的赞助者，并让我将大量的时间用在了这个社区上。非常感谢！

感谢 Erik 创建了 Rx、教会我很多东西，并为本书作序。

来自托马什的致谢

首先，我要感谢我的父母，大约 20 年前他们给了我第一台计算机（带有 8 MB 内存的 486DX2，我永远也不会忘记）。我的编程之旅就是这样开始的。有很多人为这本书的撰写做出了贡献。首先是本，他同意编写本书的第 1 章和最后一章，并审阅了我编写的内容。

提到审阅者，Venkat Subramaniam 花费了很多精力，让我能够用一种恰当且一致的方式来组织本书内容。他经常建议我使用不同的句子、段落和章节顺序组织内容，甚至建议删除整页不相关的内容。我们另外一位审阅者是极有见识和经验的 Dávid Karnok。作为 RxJava 的项目负责人，他发现了许多 bug、竞态条件、不一致性和其他问题。这两位审阅者都提供了数百条评论，极大地提高了这本书的质量。在这本书的最初阶段，我的许多同事阅读了手稿，并给出了非常有价值的反馈。我要感谢 Dariusz Baciński、Szymon Homa、Piotr Pietrzak、Jakub Pilimon、Adam Wojszczyk、Marcin Zajączkowski 和 Maciej Ziarko。

电子书

扫描如下二维码，即可购买本书中文电子版。

第 1 章

使用RxJava实现反应式编程

本 · 克里斯滕森（Ben Christensen）

RxJava 是对 Java 和 Android 进行反应式编程的具体实现，它受到了函数式编程的影响。RxJava 倡导函数组合，避免出现全局状态和副作用，并且要以流的方式思考，进而组合异步和基于事件的程序。它起源于观察者模式（observer pattern）的生产者 / 消费者回调，并且扩展了几十个操作符来实现组合、转换、调度、节流、错误处理以及生命周期管理。

RxJava 是一个成熟的开源库，已经被服务器端和 Android 移动设备广泛采用。除了这个库之外，开发人员还围绕 RxJava 和反应式编程构建了一个活跃的社区，主要用来改进项目、互相交流、撰写文章以及提供帮助。

这一章将概述 RxJava，讨论什么是 RxJava，以及它如何运行。本书的其余部分会带你了解 RxJava 的全部细节，以及如何将其用于应用程序。你在阅读本书的时候，可以没有任何反应式编程的经验，因为本书会从头开始，带领你了解 RxJava 的理念和实践，以便将它的优势应用到具体用例中。

1.1　反应式编程与RxJava

反应式编程（reactive programming）是一个通用的编程术语，它主要关注对变更做出反应，比如数据值或事件。反应式编程通常可以按照命令式（imperative）的方式来实现。回调就是一种以命令式实现反应式编程的方法。电子表格是反应式编程的一个绝佳例子：某些单元格依赖于其他的单元格，如果被依赖的单元格发生变化，这些单元格也会随之“做出反应”。

函数式反应编程？

尽管 Reactive Extensions（通常指 Rx，特指的话是 RxJava）受到了函数式编程的影响，但它并不是函数式反应编程（Functional Reactive Programming，FRP）。FRP 是非常具体的一种反应式编程类型，它涉及连续的时间，而 RxJava 只处理随时间推移出现的离散事件。在 RxJava 的早期，我本人也陷入了这样的命名陷阱，将其宣传为“函数式反应”，后来我发现“函数式反应”在数年前就已经被别人定义了。因此除了“反应式编程”之外，没有一个公认的通用术语来描述 RxJava。FRP 还是经常被误用于描述 RxJava 和类似的方案。对于是应该拓展 FRP 的含义（因为在过去的几年间，它已经被非正式地使用了），还是让 FRP 止于关注连续时间的实现，互联网上也时有争论。

为了消除疑虑，可以把重点放在 RxJava 确实受到了函数式编程的影响，并且有意地采取了与命令式编程不同的编程模型。这一章提到“反应式”时，指的是 RxJava 使用的反应式 + 函数式风格。与之相对，提到“命令式”时，并不是说反应式编程不能以命令式的方式实现，强调的是使用命令式的方式来编程，而不是 RxJava 的函数式风格。专门对比命令式方式和函数式方式时，为了准确起见，会使用“反应式 – 函数式”和“反应式 – 命令式”。

在现在的计算机中，当涉及操作系统和硬件时，一切都会变成命令式的。开发人员必须明确告诉计算机要完成什么以及如何实现。人类不会像 CPU 和相关系统那样思考，所以我们添加了抽象。反应式 – 函数式编程就是一种抽象，就像高层级命令式编程术语是对底层二进制和汇编指令的抽象一样。记住并理解“一切都会变成命令式的”很重要，因为它能够帮助理解反应式 – 函数式编程的思维模型，并理解它最终是如何执行的，这里并没有什么魔法。

因此，作为一种编程方式，反应式 – 函数式编程是命令式系统之上的一种抽象。它允许开发人员在编写异步和事件驱动的用例时不用像计算机本身那样思考，也不用以命令式的方式来定义复杂的状态交互，尤其是跨线程和网络边界时。在处理异步和事件驱动的系统时，不用像计算机那样思考是一项有用的特质，因为这种情况会涉及并发和并行，而要正确和高效地使用这些功能是非常具有挑战性的。Brian Goetz 的著作《Java 并发编程实战》、Doug Lea 的著作 *Concurrent Programming in Java* 以及像 Mechanical Sympathy 这样的论坛，都表明了掌握并发所面临的深度、广度以及复杂性。使用 RxJava 以来，通过与这些书的作者、论坛和社区的专家的交流，我更加确信编写高性能、高效、可扩展和正确处理并发的软件相当不容易。这还没有将分布式系统考虑进来呢，它将并发性和并行性的难度提高了一截。

所以，简而言之，反应式 – 函数式编程解决的问题就是并发和并行。更通俗地说，它解决了回调地狱问题。回调地狱是以命令式的方式来处理反应式和异步用例带来的问题。反应式编程，比如 RxJava 实现，受到了函数式编程的影响，并且会使用声明式的方式来避免反应式 – 命令式代码常见的问题。

1.2 何时需要反应式编程

反应式编程在如下场景中非常有用。

- 处理用户事件，比如鼠标移动和单击、键盘输入、GPS 信号因用户设备的移动而不断变化、设备陀螺仪信号和触摸事件等。
- 响应和处理来自磁盘或网络的所有延迟受限的 IO 事件，IO 本质上是异步的（发起请求，时间推移，可能收到也可能收不到响应，触发下一步事件）。
- 在应用程序中处理由该应用程序无法控制的生产者推送过来的事件或数据（来自服务器的系统事件、上述用户事件、来自硬件的信号、模拟世界中由传感器触发的事件等）。

如果涉及的代码只处理一个事件流，那么使用带有回调的反应式 – 命令式编程就很好，引入反应式 – 函数式编程并不会带来太多的收益。如果你有数百个不同的事件流，而且它们彼此独立，命令式编程也不会有太大的问题。在这种直接的使用场景中，命令式是最高效的方式，因为它消除了反应式编程的抽象层，并且更加契合对当前操作系统、语言和编译器的优化。

如果你的程序像大多数程序一样，那么你需要组合事件（或者函数或网络调用的异步响应）、包含事件交互的条件逻辑，而且在所有调用之后必须处理故障场景和清理资源。在这种情况下，反应式 – 命令式的复杂性会急剧增加，而反应式 – 函数式编程则能体现出它的价值了。我认同一个未经科学验证的观点，那就是反应式 – 函数式编程难入门而且学习曲线较陡峭，但是它的复杂性要远远低于反应式 – 命令式编程。

这就是称 Reactive Extensions（Rx）和 RxJava 是“用于组合异步和基于事件的程序的库”的原因。RxJava 是反应式编程原则的具体实现，受到了函数式以及数据流编程的影响。其实，我们有不同的方式来实现“反应式”，RxJava 只是其中之一。接下来深入研究一下它是如何运行的。

1.3 RxJava是如何运行的

RxJava 的核心是 `Observable` 类型，它代表了数据或事件的流。它的目的是实现推送（反应式），但是也可以用于拉取（交互式）。它是延迟执行的（lazy），不是立即执行的（eager）。它可以同步使用，也可以异步使用。它能够代表随着时间推移产生的 0 个、1 个、多个或者无穷个值或事件。

这涉及很多的术语和细节，需要一一介绍，2.1 节将介绍完整的细节。

1.3.1 推送与拉取

RxJava 实现反应式的要点在于它支持推送，所以 `Observable` 和关联的 `Observer` 类型签名支持把事件推送给它。这通常会伴随着异步，1.3.2 节会进行讨论。但是，`Observable` 类型还支持一个异步的反馈通道（有时也称为异步 – 拉取或反应式拉取），作为异步系统中的一种流控制或回压方式。本章后面会讨论流控制，以及如何使用该机制。

为了支持接收推送来的事件，`Observable`/`Observer` 通过订阅进行连接。`Observable` 代表了数据流，它可以被 `Observer` 订阅（2.2 节会介绍更多内容）。

```
interface Observable<T> {
    Subscription subscribe(Observer s)
}
```

订阅之后，Observer 就能够接收三种推送给它的事件。

- 通过 onNext() 函数推送的数据。
- 通过 onError() 函数推送的错误（异常或 Throwable）。
- 通过 onCompleted() 函数推送的流完成信息。

```
interface Observer<T> {
    void onNext(T t)
    void onError(Throwable t)
    void onCompleted()
}
```

其中，onNext() 可能永远也不会被调用，也可能会被调用一次、多次或无数次。onError() 和 onCompleted() 是终端事件，这意味着两者只能有一个被调用，并且只能被调用一次。终端事件被调用之后，Observable 流就完成了，以后就不能再向它发送事件了。如果流是无限的，并且没有发生故障，那么终端事件可能永远不会发生。

6.1 节和 6.2 节将会展示另外一种类型签名，它支持交互式拉取。

```
interface Producer {
    void request(long n)
}
```

它可以与更加高级的 Observer 协同使用，即 Subscriber（2.3 节提供了更多的细节）。

```
abstract class Subscriber<T> implements Observer<T>, Subscription {
    void onNext(T t)
    void onError(Throwable t)
    void onCompleted()
    ...
    void unsubscribe()
    void setProducer(Producer p)
}
```

Subscription 接口中包含了 unsubcribe 函数，该函数能够允许订阅者取消对某个 Observable 流的订阅。setProducer 函数和 Producer 类型用来在生产者和消费者之间建立一个双向的通信通道，该通道用于流控制。

1.3.2 异步与同步

一般而言，Observable 是异步的，但它并非总是如此。Observable 可以是同步的，事实上，它默认就是同步的。除非要求，否则 RxJava 永远不会添加并发功能。同步的 Observable 将被订阅，使用订阅者的线程发布（emit）所有数据并且完成（如果是有限 Observable 的话）。由阻塞式网络 I/O 支撑的 Observable 将会同步阻塞订阅线程，并在阻塞网络 I/O 返回时，通过 onNext() 发布数据。

例如，以下代码示例完全是同步的。

```
Observable.create(s -> {
    s.onNext("Hello World!");
    s.onCompleted();
}).subscribe(hello -> System.out.println(hello));
```

2.2 节和 2.4.1 节将介绍 `Observable.create` 和 `Observable.subscribe` 的更多知识。

现在，你可能会想，这并不是反应式系统的理想行为。你是对的！将同步阻塞 I/O 与 `Observable` 组合使用是一种很糟糕的形式（如果确实要使用阻塞 I/O 的话，它需要以线程的方式进行异步化）。但是，有时候从内存缓存中同步获取数据并立即返回也是一种恰当的做法。前面的“Hello World”样例并不需要并发性，事实上，如果为其添加异步调度的话，它将会慢得多。因此，通常来讲，实际上重要的标准是 `Observable` 生成事件的过程是阻塞的还是非阻塞的，而非它是同步的还是异步的。“Hello World”样例是非阻塞的，因为它永远不会阻塞线程，所以使用 `Observable` 是正确的（尽管看上去有些多余）。

实际上，RxJava 的 `Observable` 不知道异步与同步，也不知道并发性是否存在以及来自何处。设计就是如此，这允许 `Observable` 的实现来决定什么做法是最好的。为什么说这是有用的呢？

首先，并发性可以来自多个地方，而不仅仅是线程池。如果数据源已经借助事件循环（event loop）实现了异步，那么 RxJava 不应该添加更多的调度开销或者强制使用特定的调度实现。并发性可以来自线程池、事件循环、Actor 等。它可能是手动添加的，也可能来源于数据源。异步性来自何处，RxJava 并不知晓。

其次，使用同步的行为有两个很好的理由，下面会进行阐述。

1. 内存数据

如果数据在本地内存缓存中（查询时间固定在毫秒 / 纳秒级别），那么再花费调度成本将其异步化就没有意义了。`Observable` 可以同步地获取数据，并将其发布到订阅线程上，如下所示。

```
Observable.create(s -> {
    s.onNext(cache.get(SOME_KEY));
    s.onCompleted();
}).subscribe(value -> System.out.println(value));
```

不清楚数据是否在内存中的时候，调度选择是很重要的。如果数据在内存中，就采用同步的方式进行发布；如果不在内存中，就执行异步的网络调用，并在数据到达的时候将其返回。这种选择可以放到一个条件化的 `Observable` 中。

```
//伪代码
Observable.create(s -> {
    T fromCache = getFromCache(SOME_KEY);
    if(fromCache != null) {
        //同步发布
        s.onNext(fromCache);
        s.onCompleted();
    } else {
        //异步抓取
        getDataAsynchronously(SOME_KEY)
            .onResponse(v -> {
                putInCache(SOME_KEY, v);
                s.onNext(v);
                s.onCompleted();
            })
```

```
                .onFailure(exception -> {
                    s.onError(exception);
                });
        }
}).subscribe(s -> System.out.println(s));
```

2. 同步计算（如操作符）

保持同步的更常见原因是通过操作符进行流组合和转换。RxJava 会使用大量的操作符 API 来操作、组合和转换数据，比如 `map()`、`filter()`、`take()`、`flatMap()` 和 `groupBy()`。大多数这样的操作符是同步的，这意味着在事件经过的时候，它们会在 `onNext()` 中执行同步计算。

出于性能原因，这些操作符都是同步的。以下面的代码为例。

```
Observable<Integer> o = Observable.create(s -> {
    s.onNext(1);
    s.onNext(2);
    s.onNext(3);
    s.onCompleted();
});

o.map(i -> "Number " + i)
 .subscribe(s -> System.out.println(s));
```

假如 `map` 操作符默认是异步的，(1, 2, 3) 中的每个数字都会调度到一个线程上，在这个线程上将会执行字符串连接（`"Number " + i`）。这是非常低效的，调度、上下文切换等一般还会造成不确定的延迟。

这里需要着重理解的就是大多数的 `Observable` 函数管道是同步的（除非某个特定的操作符需要是异步的，比如 `timeout` 或 `observeOn`），而 `Observable` 本身可以是异步的。这些主题在 4.9.5 节和 7.1.3 节会深入介绍。

如下的样例展现了同步和异步的混合使用。

```
Observable.create(s -> {
    ... async subscription and data emission ...
})
.doOnNext(i -> System.out.println(Thread.currentThread()))
.filter(i -> i % 2 == 0)
.map(i -> "Value " + i + " processed on " + Thread.currentThread())
.subscribe(s -> System.out.println("SOME VALUE =>" + s));
System.out.println("Will print BEFORE values are emitted")
```

本例中的 `Observable` 是异步的（它会在与订阅者不同的线程中发布事件），所以订阅是非阻塞的，最后的 `println` 的输出将会早于事件的传播，并先于“SOME VALUE ⇒”显示。

但是，`filter()` 和 `map()` 函数是同步执行的，它们会在调用发布事件的线程中执行。一般来说，这是期望的行为：实现异步的管道（`Observable` 和组合操作符），让事件保持高效的同步计算。

因此，`Observable` 类型本身支持同步和异步的具体实现，其设计本意即是如此。

1.3.3 并发与并行

单个的 Observable 流既不允许并发，也不允许并行。相反，它们是通过组合异步 Observable 来实现的。

并行（parallelism）指的是任务同时执行，通常会在不同的 CPU 或机器上。而并发（concurrency）指的是多任务的组合和交叉。如果一个 CPU 上面有多个任务的话（比如线程），它们是通过“时间分片”实现的并发，而不是并行。每个线程会得到一部分的 CPU 时间，然后即便该线程尚未完成，也要将 CPU 时间让给其他线程。

根据定义，并行执行是并发的，但是并发不一定是并行的。实际上，这意味着多线程是并发的，但是只有这些线程同时被调度到不同的 CPU 上并在其上执行时，才是并行。因此，通常我们会讨论并发性和如何并发，并行则是一种特定形式的并发。

RxJava 的 Observable 契约要求事件（onNext()、onCompleted()、onError()）始终避免并发发布。换句话说，单个 Observable 流必须始终是序列化和线程安全的。每个事件可以从不同的线程中发布出来，只要发布不是并发的即可。这意味着 onNext() 没有交叉或同时执行。如果 onNext() 依然还在某个线程上执行，那么其他的线程将不能再次调用它（交叉）。

下面的样例展现了正确的代码。

```
Observable.create(s -> {
  new Thread(() -> {
    s.onNext("one");
    s.onNext("two");
    s.onNext("three");
    s.onNext("four");
    s.onCompleted();
  }).start();
});
```

这段代码顺序地发布数据，所以它符合契约。（注意，一般不建议这样在 Observable 中启动线程，而应使用调度器，参见 4.9 节中的讨论。）

如下的样例展现了非法代码。

```
//不要这样做
Observable.create(s -> {
  //线程A
  new Thread(() -> {
    s.onNext("one");
    s.onNext("two");
  }).start();

  //线程B
  new Thread(() -> {
    s.onNext("three");
    s.onNext("four");
  }).start();

  //由于线程竞争，不需要发布s.onCompleted()
});
//不要这样做
```

这段代码是非法的，因为它的两个线程能够并发地调用 onNext()，这破坏了契约。（同时，它还需要安全地等待两个线程都完成才能安全地调用 onComplete，如前所述，这样手动启动线程很糟糕。）

那么，该如何结合 RxJava 发挥并发和并行的优势呢？那就是组合。

单个 Observable 流始终是序列化的，但是每个 Observable 可以独立于其他 Observable 来进行操作，因此能够实现并发 / 并行。这也是 merge 和 flatMap 在 RxJava 中如此常用的原因所在——并发地将异步流组合在一起（参见 3.1.2 节和 3.2.1 节）。

如下是编造的一个例子，展示了在各自线程中运行异步 Observable 以及将其合并起来的机制。

```
Observable<String> a = Observable.create(s -> {
  new Thread(() -> {
    s.onNext("one");
    s.onNext("two");
    s.onCompleted();
  }).start();
});

Observable<String> b = Observable.create(s -> {
  new Thread(() -> {
    s.onNext("three");
    s.onNext("four");
    s.onCompleted();
  }).start();
});

//并发订阅a和b，并将它们合并到第三个序列化流中
Observable<String> c = Observable.merge(a, b);
```

Observable c 将会收到来自 a 和 b 的条目，因为它们的异步性，将会发生以下三件事。

- one 会出现在 two 之前。
- three 会出现在 four 之前。
- one/two 和 three/four 的顺序是不确定的。

那么，到底为什么不允许并发地调用 onNext() 呢？

主要原因在于 onNext() 本意是供人类使用的，并发比较困难。假如能够并发调用 onNext()，这就意味着所有 Observer 都需要为并发调用进行防御性编码，即便本来并不期望或想要这样调用。

第二个原因在于有些操作无法实现并发发布，比如 scan 和 reduce，它们是非常常见且重要的行为。像 scan 和 reduce 这样的操作符需要有顺序的事件传播，这样状态才能在事件流上累积，这些事件不能兼具组合性（associative）和可交换性（commutative）。允许并发的 Observable 流（具有并发的 onNext()）将限制能够处理的事件类型，并且需要线程安全的数据结构。

> Java 8 的 Stream 类型支持并发发布。这也是 java.util.stream.Stream 需要 reduce 函数具备组合性的原因，它们必须支持在并行流上并发调用。在 java.util.stream 包的文档中，关于并行、排序（与交换性相关）、reduction 操作以及组合性的部分进一步阐述了相同 Stream 类型允许顺序发布和并发发布的复杂性。

第三个原因在于同步开销会影响性能，因为所有的 Observer 和操作符都需要是线程安全的，即便大多数情况下数据是顺序到达的。尽管 JVM 通常擅长消除同步开销，但是并非在所有的场景中都如此（尤其是使用原子化的非阻塞算法时更是如此），最终导致顺序流上产生不必要的性能开销。

除此之外，进行一般的细粒度并行通常会更慢。并行一般需要在较粗的粒度上进行，比如批处理工作，以弥补切换线程、调度工作和重新组合的开销。如果在单个线程上同步执行，并充分利用针对有顺序的计算的内存和 CPU 优化，那么将会高效得多。对于 List 或 array 来说，为批处理并行找到合适的默认做法是很容易的，因为所有的条目都是提前可知的，并且能够划分为批处理（但是即便如此，通常来讲在一个 CPU 上处理整个列表也会更快，除非列表非常庞大或者对每个条目的处理非常耗时）。但是，流无法预先了解工作的情况，只能通过 onNext() 接收数据，因此无法自动对工作进行分块。

实际上，在 v1 版本之前，RxJava 添加过一个 .parallel(Function f) 操作符，它的行为类似于 java.util.stream.Stream.parallel()，当时认为这是非常便利的。它的实现方式并没有打破 RxJava 的契约，首先将一个 Observable 分割为多个并行执行的 Observable，随后再将它们合并到一起。但是它在该库更新到 v1 版本之前被删除了，因为它令人费解，并且总是会降低性能。为事件流添加并行计算通常都需要进行推理和测试。也许，ParallelObservable 能够提供一些帮助，为此操作符被限定在具备结合性的子集中。但是在使用 RxJava 的时代并不值得尝试，因为组合使用 merge 和 flatMap 就是针对这种用案的有效构造。

第 3 章将会讲解如何使用操作符组合 Observable，使其能够从并发和并行中受益。

1.3.4　延迟执行与立即执行

Observable 类型是**延迟执行**的，这意味着在订阅它之前，它什么事情都不会做。这与**立即执行**类型有所不同，比如 Future，它在创建之时就开始活跃工作了。延迟执行允许组合 Observable，不会因为竞态条件时缺乏缓存而丢失数据。在 Future 中，这并不是什么问题，因为单个值可以缓存起来，所以如果值在组合之前就已经传递到了，值依然能够被获取。在一个无界的流中，需要无界的缓冲提供同样的保证。因此，Observable 是延迟执行的，只有被订阅之后才会启动，所在在数据开始流动之前可以完成所有组合。

实际上，这意味着两件事。

订阅之前，所有的构造都不会开始工作。

因为 Observable 延迟执行的特性，创建之后其实它并不会开启任何工作（请忽略分配 Observable 对象本身这项“工作”）。它只是定义在它最终被订阅时，应该完成什么工作。请考虑如下定义的 Observable。

```
Observable<T> someData = Observable.create(s -> {
    getDataFromServerWithCallback(args, data -> {
        s.onNext(data);
        s.onCompleted();
    });
})
```

现在已经存在 someData 引用，但是 getDataFromServerWithCallback 还没有执行。这里所发生的就是 Observable 包装器声明了一组要执行的工作，也就是 Observable 中的函数。

订阅 Observable 会让任务开始执行。

```
someData.subscribe(s -> System.out.println(s));
```

这样的话，就会延迟执行 Observable 代表的工作。

Observable 可以重用。

因为 Observable 是延迟执行的，也就意味着一个特定的实例可以调用多次。继续分析前面的样例，这意味着我们可以这样做：

```
someData.subscribe(s -> System.out.println("Subscriber 1: " + s));
someData.subscribe(s -> System.out.println("Subscriber 2: " + s));
```

现在有了两个单独的订阅，每个都调用 getDataFromServerWithCallback 并发布事件。

延迟执行不同于其他的异步类型，比如 Future，创建的 Future 代表了已经开始的工作。Future 不能重用（订阅多次来触发工作）。如果存在对 Future 的引用，那么就意味着工作已经开始执行。前面的示例代码展示了立即执行是什么。getDataFromServerWithCallback 就是立即执行的，因为在调用的时候，它会马上执行。围绕 getDataFromServerWithCallback 包装一个 Observable，就能够以延迟执行的形式使用它。

在进行组合的时候，延迟执行是非常有用的，如下所示。

```
someData
    .onErrorResumeNext(lazyFallback)
    .subscribe(s -> System.out.println(s));
```

在这个样例中，lazyFallback Observable 代表了将来可以完成的工作。但是这些工作只在有人订阅的时候才会完成，而我们想要只在 someData 失败的时候才订阅它。当然，立即执行类型可以通过使用函数调用实现延迟执行（比如 getDataAsFutureA()）。

虽然延迟执行和立即执行各有优劣，但是 RxJava 中的 Observable 是延迟执行的。因此 Observable 在被订阅之前，它是不会做任何事情的。

4.3 节还会详细讨论这个话题。

1.3.5　双重性

Rx 的 Observable 是一个异步的“双重”（dual）Iterable。所谓“双重”，指的是 Observable 提供了 Iterable 的所有功能，但数据方向相反：它推送数据，而不是拉取数据。表 1-1 展现了提供推送和拉取功能的类型。

表1-1：提供推送和拉取功能的类型

拉取（Iterable）	推送（Observable）
T next()	onNext(T)
抛出异常	onError(Throwable)
返回	onCompleted()

从表 1-1 可以看出，数据不是由消费者通过 next() 拉取的，而是由生产者通过 onNext(T) 推送的。成功的终止要通过 onCompleted() 回调进行标记，而不是阻塞线程直到遍历完所有的条目。在出现错误时，也不是在调用栈上抛出异常，而是将错误以事件的形式发布到 onError(Throwable) 回调上。

它的行为具有双重性意味着通过 Iterable 和 Iterator 实现的同步拉取，都能通过 Observable 和 Observer 以异步推送的方式来实现。这样的话，相同的编程模型可以应用于这两者。

例如，Java 8 中的 Iterable 可以通过 java.util.stream.Stream 类型实现函数组合，如下所示。

```
//将包含75个字符串的Iterable作为 Stream
getDataFromLocalMemorySynchronously()
    .skip(10)
    .limit(5)
    .map(s -> s + "_transformed")
    .forEach(System.out::println)
```

此例将会从 getDataFromLocalMemorySynchronously() 中检索 75 个字符串，并且只获取顺序从 11 到 15 的字符串，忽略其他字符串，然后转换获取到的字符串并将其打印出来。（3.4 节将会介绍更多的操作符，如 take、skip 和 limit。）

RxJava Observable 的使用方式与之相同。

```
//会发布75个字符串的Observable
getDataFromNetworkAsynchronously()
    .skip(10)
    .take(5)
    .map(s -> s + "_transformed")
    .subscribe(System.out::println)
```

此例将会接收 5 个字符串（发布了 15 个字符串，但是前 10 个丢弃了），然后取消订阅（忽略或停止要发布的其他字符串）。它会进行转换并将字符串打印出来，与前面的 Iterable/Stream 样例类似。

换句话说，Rx 中的 Observable 允许通过推送的方式对异步数据进行编程，就像 Streams 围绕 Iterable 和 List 以同步拉取的方式进行编程一样。

1.3.6 基数

Observable 支持异步推送多个值。这很好地匹配表 1-2 中的右下角，具有异步双重性的 Iterable（或 Stream、List、Enumerable 等）以及多值版本的 Future。

表1-2：基数

	一　　个	多　　个
同步	T getData()	Iterable<T> getData()
异步	Future<T> getData()	Observable<T> getData()

注意，这一节指的是通用的 Future，它使用 Future.onSuccess(callback) 语法来代表它的行为。存在各种实现，比如 CompletableFuture、ListenableFuture 或 Scala 的 Future。但是不管使用哪一个实现，切记不要使用 java.util.Future，因为它在获取值的时候需要阻塞。

那么，为什么 Observable 比单纯的 Future 更有价值呢？最明显的原因是 Observable 能够处理一个事件流或多值响应。而另外一个不那么显而易见的原因在于，它能够对多个具有单个值的响应进行组合。下面依次来了解一下。

1. 事件流

事件流非常简单。随着时间的推移，生产者将事件推送给消费者，如下所示。

```
//生产者
Observable<Event> mouseEvents = ...;

//消费者
mouseEvents.subscribe(e -> doSomethingWithEvent(e));
```

如果使用 Future 的话，它将无法很好地运行。

```
//生产者
Future<Event> mouseEvents = ...;

//消费者
mouseEvents.onSuccess(e -> doSomethingWithEvent(e));
```

onSuccess 回调可能会收到“最后的事件”，但是有些问题依然存在：消费者现在是否需要轮询？生产者是否要对它们进行排队？或者，在每次获取之间，它们是否会丢失？Observable 在这里肯定是有帮助的。而在没有 Observable 的情况下，回调方法要比将其建模为 Future 更好。

2. 多个值

多值响应是 Observable 的另一个用武之地。一般来讲，任何使用 List、Iterable 或 Stream 的地方，都可以使用 Observable 来替换。

```
//生产者
Observable<Friend> friends = ...

//消费者
friends.subscribe(friend -> sayHello(friend));
```

现在，也可以和 Future 协作使用，如下所示。

```
//生产者
Future<List<Friend>> friends = ...

//消费者
```

```
friends.onSuccess(listOfFriends -> {
    listOfFriends.forEach(friend -> sayHello(friend));
});
```

那么，为什么要使用 Observable<Friend> 方式呢？

如果返回的数据列表比较小的话，这可能对性能并没有太大的影响，选择哪种方式完全取决于个人喜好。但是，如果列表非常大，或者远程数据源必须从不同的位置获取列表的不同部分，那么 Observable<Friend> 方式在性能和延迟性方面会有更好的表现。

最让人信服的原因就是，可以在接收到条目的时候就进行处理，而不必等到整个集合的内容全部抵达再进行处理。如果后端的各种网络延迟会对每个条目造成不同影响的话，这一点就更为明显，因为长尾延迟（比如在面向服务的架构或微服务架构中）和共享数据存储，这种情况确实很常见。如果要等待整个集合的话，消费者将会经历完成该集合聚合需要的最长延迟。如果条目是以 Observable 流的形式返回，那么消费者能够立即接收它们，"第一个条目的耗时"可能会明显低于最后一个和最慢条目的耗时。要使这种方式能够正常运行，必须要牺牲流的顺序，这样才能让服务器按照收到的顺序发布条目。如果顺序对用户来说非常重要，那么在条目数据或元数据中可以包含一个排序或位置信息，这样客户端可以按需对条目进行排序和定位。

此外，这种方式还能按每个条目的需求分配内存，而不必为整个集合分配和收集内存。

3. 组合

在组合单值响应，比如组合来自 Future 的响应时，多值的 Observable 类型也是非常有用的。

在合并多个 Future 时，需要发布另外一个带有单个值的 Future，如下所示。

```
CompletableFuture<String> f1 = getDataAsFuture(1);
CompletableFuture<String> f2 = getDataAsFuture(2);

CompletableFuture<String> f3 = f1.thenCombine(f2, (x, y) -> {
  return x+y;
});
```

这可能就是想要的效果，实际上也可以通过 RxJava 的 Observable.zip 来实现（3.2.2 节将会介绍更多这方面的知识）。

```
Observable<String> o1 = getDataAsObservable(1);
Observable<String> o2 = getDataAsObservable(2);

Observable<String> o3 = Observable.zip(o1, o2, (x, y) -> {
  return x+y;
});
```

但是，这意味着在发布内容之前必须要等待，直到所有 Future 完成。通常，更好的方式是在每个任务完成的时候就发布 Future 返回的值。本例使用 Observable.merge（或相关的 flatMap）会是更好的方案。它允许将结果（即便每个结果都是发布一个值的 Observable）组合为由值组成的流，这些值在它们就绪的时候就会发布出来。

```
Observable<String> o1 = getDataAsObservable(1);
Observable<String> o2 = getDataAsObservable(2);
```

```
//现在，o3是o1和o2组成的流，会立即发布条目，无须等待
Observable<String> o3 = Observable.merge(o1, o2);
```

4. Single

现在来看，尽管 Rx Observable 在处理多值流的时候非常棒，但简洁的单值表示非常适合 API 的设计和使用。此外，基本的请求 / 响应行为在应用程序中非常普遍。基于此，RxJava 提供了一个 Single 类型，它与 Future 是对等的，只不过它是延迟执行的。可以将 Single 视为具备两个特殊优点的 Future：首先，它是延迟执行的，所以它可以被多次订阅并且易于组合；其次，它适配 RxJava 的 API，所以它能够很容易地与 Observable 交互。

例如，考虑如下的访问器。

```
public static Single<String> getDataA() {
    return Single.<String> create(o -> {
      o.onSuccess("DataA");
    }).subscribeOn(Schedulers.io());
}

public static Single<String> getDataB() {
    return Single.just("DataB")
            .subscribeOn(Schedulers.io());
}
```

它们还可以按照下面的方式使用和组合。

```
//将a和b合并到由两个值组成的Observable流中
Observable<String> a_merge_b = getDataA().mergeWith(getDataB());
```

请注意这两个 Single 是如何合并到一个 Observable 中的。这样发布出的数据可能是 [A, B]，也可能是 [B, A]，具体取决于哪个先完成。

回到前面的例子，现在可以使用 Single 替换 Observable 来表示数据的获取，但是要将它们合并到一个值的流。

```
//Observable<String> o1 = getDataAsObservable(1);
//Observable<String> o2 = getDataAsObservable(2);

Single<String> s1 = getDataAsSingle(1);
Single<String> s2 = getDataAsSingle(2);

//现在，o3是s1和s2组成的流，会立即发布条目，无须等待
Observable<String> o3 = Single.merge(s1, s2);
```

使用 Single 来代替 Observable 表示“单个值的流”能够简化使用，因为开发人员必须要考虑 Single 只能具有如下行为中的一种。

- 响应错误。
- 永远不响应。
- 响应成功。

与之相对，消费者在使用 Observable 的时候，必须要考虑一些其他的状态。

- 响应错误。
- 永远不响应。

- 响应成功，没有数据并终止。
- 响应成功，有单个数据值并终止。
- 响应成功，有多个数据值并终止。
- 响应成功，有一个或多个值，并且永远不终止（等待更多数据）。

使用 `Single`，消费该 API 的思维模型会更加简单。只有在组合到 `Observable` 中之后，开发人员才必须考虑其他状态。这通常是它更适用的地方，因为通常由开发人员控制代码，而数据 API 通常来自第三方。

5.5 节将会介绍 `Single` 的更多知识。

5. Completable

除了 `Single` 之外，RxJava 还有一个 `Completable` 类型。它解决了没有返回类型，只需要表示成功或失败的常见场景。通常会使用 `Observable<Void>` 或 `Single<Void>`，但这有些牵强，因此才诞生了 `Completable`，如下所示。

```
Completable c = writeToDatabase("data");
```

这种场景在进行异步写入的时候很常见，此时不需要返回值，只需要通知成功或失败即可。前面使用 `Completable` 的代码类似于：

```
Observable<Void> c = writeToDatabase("data");
```

`Completable` 本身是两个回调的抽象，即成功和失败，如下所示。

```
static Completable writeToDatabase(Object data) {
  return Completable.create(s -> {
    doAsyncWrite(data,
        //成功完成的回调
        () -> s.onCompleted(),
        //抛出Throwable而失败的回调
        error -> s.onError(error));
  });
}
```

6. 零到无穷

`Observable` 能够支持的基数从零到无穷（2.4.2 节还会进一步探讨）。但是为了简单和清晰，`Single` 是“只有一个条目的 `Observable`”，而 `Completable` 则是“没有条目的 `Observable`”。

加上这些新引入的类型后，如表 1-3 所示。

表1-3：Observable能够支持的基数从零到无穷

	零　个	一　个	多　个
同步	`void doSomething()`	`T getData()`	`Iterable<T> getData()`
异步	`Completable doSomething()`	`Single<T> getData()`	`Observable<T> getData()`

1.4　阻塞I/O与非阻塞I/O

到目前为止，关于反应式－函数式风格编程的讨论主要在于提供一个异步回调的抽象机制，以允许实现更易管理的组合。显然，如果将不相关的网络请求并发执行而不是顺序执

行的话，将会改善用户的延迟体验，这是采用异步和需要组合的原因。

但是，有性能方面的因素促使我们采用反应式的方式（不管是命令式的还是函数式的）来执行 I/O 吗？采用非阻塞 I/O 会有什么好处吗？或者说阻塞 I/O 线程以等待单个网络请求难道不可以吗？我在 Netflix 参与的性能测试表明，与每个请求对应一个线程的阻塞 I/O 相比，采用非阻塞 I/O 和事件循环会带来客观且可测量的性能收益。这一节分析了具体原因，并提供了帮助你做出决定的数据。

追寻答案的历程

在使用 RxJava 一段时间之后，我想知道阻塞 I/O 和非阻塞 I/O（具体来讲，也就是每个请求对应一个线程与事件循环）的对比结果，但是我发现很难得到明确的答案。实际上，在研究这个话题的过程中，我发现了自相矛盾的答案、传言、理论、观点和困惑。最终，我得出了结论，那就是在理论上，所有不同方式（比如纤程、事件循环、线程和 CSP）的性能（吞吐量和延迟）应该相同，因为最终所有的方式使用的是相同的 CPU 资源。但是，在实践中，具体实现是由数据结构和算法组成的，而且必须要处理硬件的具体状况，因此，首先要“理解”硬件是如何工作的，然后掌握操作系统和运行时的实现方式。

我本人无法回答这些问题，但是我很幸运，能够和 Brendan Gregg 一起工作。毫无疑问，在这方面他有着丰富的经验。我们和 Nitesh Kant 一起工作了数月，测试 Tomcat 和基于 Netty 的应用程序。

我们特意选择了“真实”的代码，如 Tomcat 和 Netty，因为它是和生产系统的选择紧密相关的（我们已经使用了 Tomcat，并且当时正在研究使用 Netty）。它们在架构方面的主要差异在于，每个请求对应一个线程还是事件循环。

你可以在 GitHub 的 Netflix-Skunkworks/WsPerfLab 仓库中看到这项研究的细节，另外还有用于测试的代码。你还可以在 SpeakerDeck 上查阅一篇概述和演讲，题目为 *Applying Reactive Programmiing with RxJava*。

正如“追寻答案的历程”中提到的，在 Linux 上基于 Tomcat 和 Netty 对阻塞 I/O 和非阻塞 I/O 的性能进行了对比测试。这种类型的测试通常会有一定的争议性，并且难以得到非常准确的结果，所以这项测试仅适用于如下场景。

- 2015/2016 年左右的典型 Linux 系统的行为。
- Java 8（OpenJDK 和 Oracle）。
- 典型生产环境中的未经修改的 Tomcat 和 Netty。
- 具有代表性的 Web 服务请求 / 响应负载，涉及多个其他 Web 服务的组合。

基于这样的环境，得出了如下的结论。

- Netty 代码要比 Tomcat 代码更高效，并且每个请求消耗更少的 CPU。
- Netty 的事件循环架构会减少高负载情况下的线程迁移，这会提升 CPU 缓存的热度和内存的本地化，提升 CPU 每个周期的指令（Instructions-per-Cycle，IPC）数量，从而降低每个请求的 CPU 周期消耗。

• 因为 Tomcat 的线程池架构，在面临负载时 Tomcat 会有更高的延迟，这涉及服务在负载情况下的线程池锁（以及锁竞争）和线程迁移。

图 1-1 阐述了这两种架构的差异。

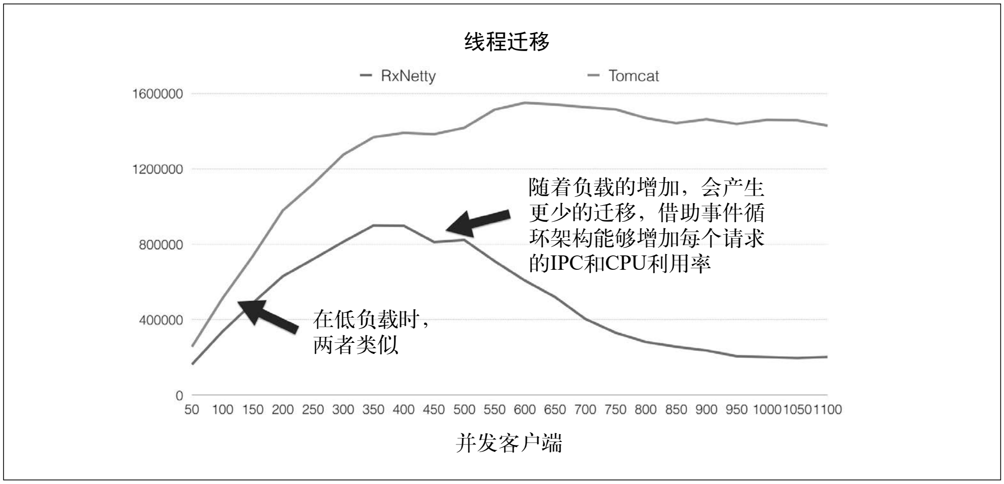


图 1-1

请注意在负载增加时这两条线的差异，它们指的是线程的迁移。在这个过程中，我发现最有意思的一件事就是，Netty 在面临负载时会更加高效，线程会变得更加“火热”并会固定到一个 CPU 核心上。而 Tomcat 则无法获得这种收益，因为每个请求都有一个单独的线程，而且因为要为每个请求调度线程，所以会导致更高的线程迁移。

如图 1-2 和图 1-3 所示，随着负载的增加，Netty 的 CPU 消耗依然近乎平稳；负载达到最大的时候，性能实际上还有轻微的提升。与之相反，Tomcat 的效率会出现下降。

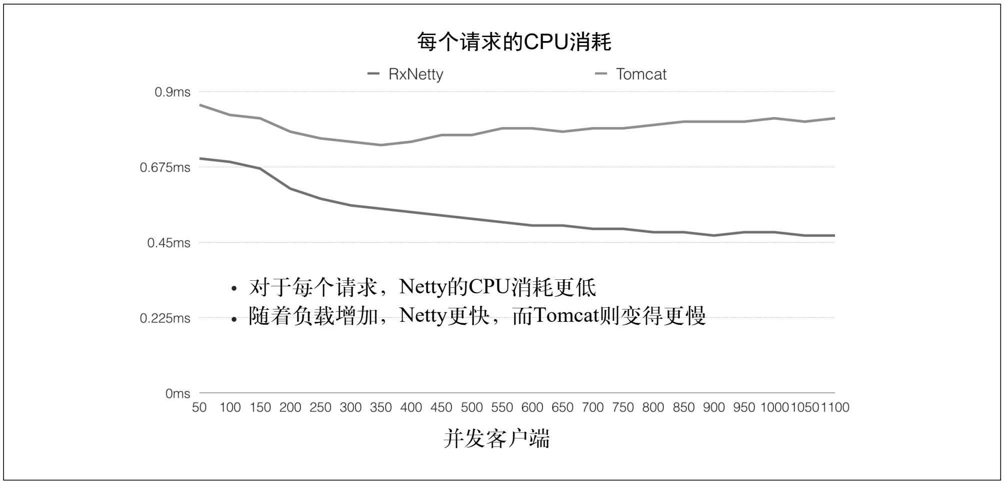


图 1-2

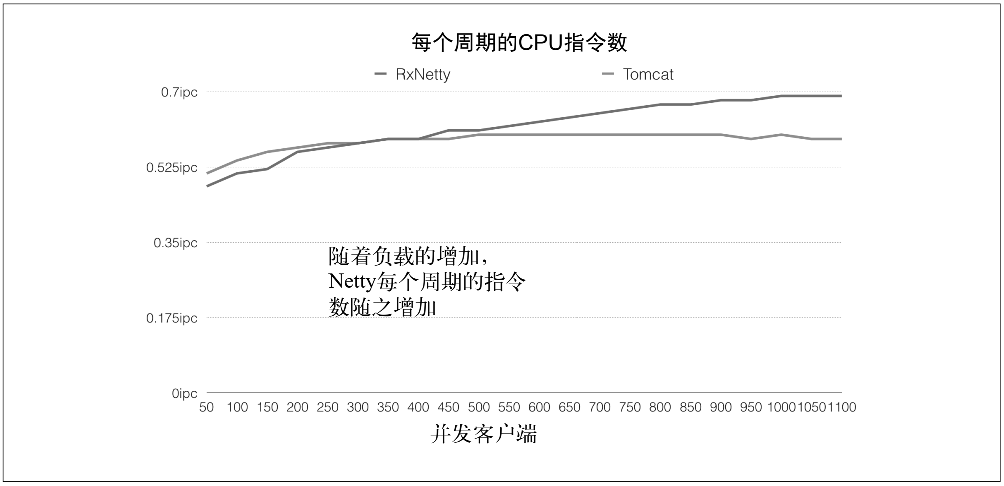


图 1-3

对延迟和吞吐量的影响如图 1-4 所示。

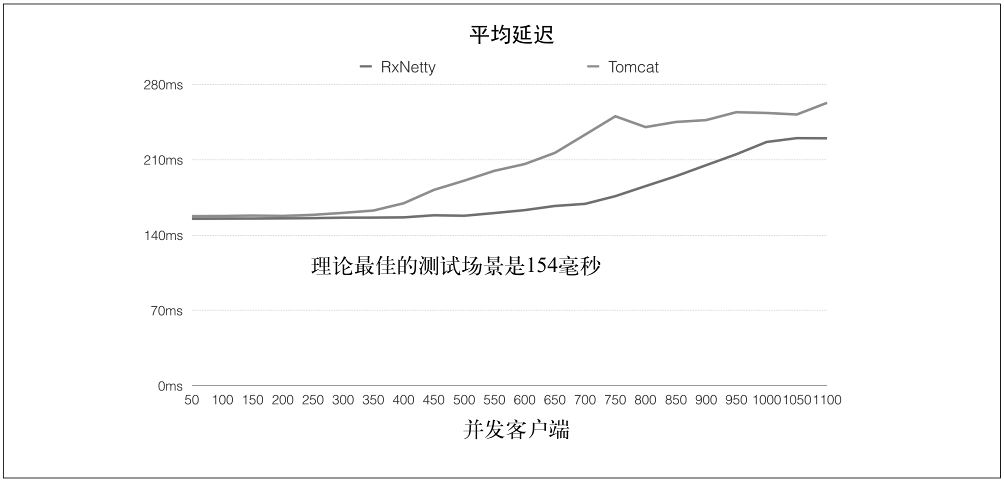


图 1-4

平均值并不是非常有价值（相对于百分位），不过从图 1-4 可以看出，在负载很低的时候，它们具有类似的延迟，但是随着负载的增加，差异变得更加明显。随着负载的增加，Netty 能够更好地利用机器的资源，对延迟的影响更小。

图 1-5 展现了异常情况如何影响用户和系统资源。Netty 能够更优雅地处理负载，避免出现最糟糕的情况。

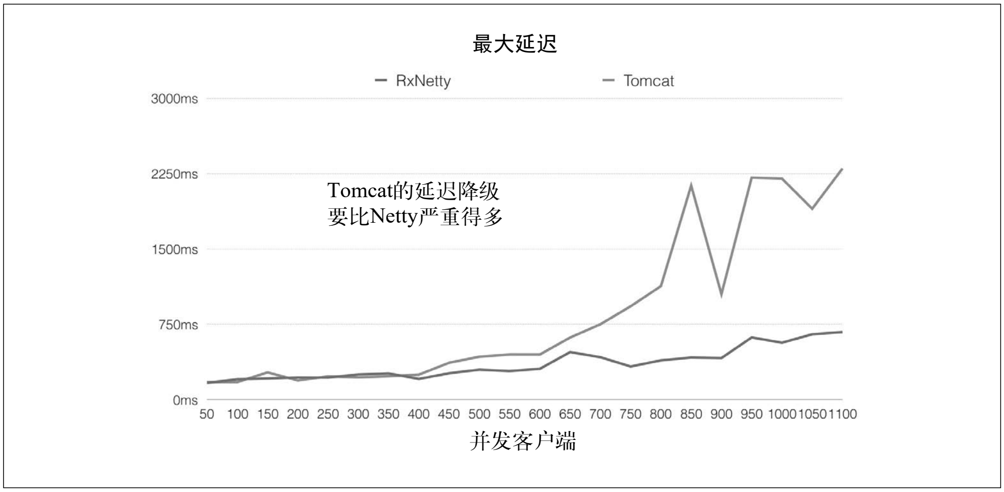


图 1-5

图 1-6 展现了吞吐量。

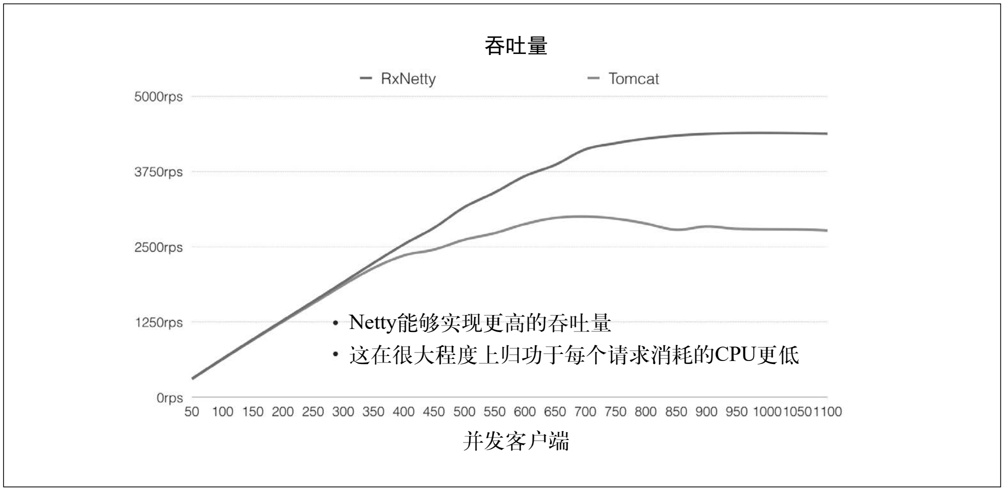


图 1-6

可以看到 Netty 方式有两个突出的优点。首先，延迟和吞吐量方面的结果更好，意味着更好的用户体验和更低的基础设施成本。其次，事件循环架构在负载情况下更加可靠。负载增加时，它并不会直接失败，反而会将机器的资源发挥到极限，并且能够更加优雅地进行处理。对于大规模生产系统来说，这非常有吸引力，因为这种系统需要处理预料之外的流量飙升，并保持系统的响应性。

我还发现事件循环架构更易于运维。为了获取最优的性能，[1] 它并不需要额外的调优，而每个请求对应一个线程的架构通常需要根据负载调整线程池的大小（以及相关的垃圾收集）。

这并不是针对该话题的详尽研究，但是这个实验和结果数据是借助非阻塞 IO 和事件循环实现“反应式”架构的有力证据。换句话说，在 2015/2016 年的硬件、Linux 内核和 JVM 上，通过事件循环进行非阻塞 I/O 确实有好处。

5.1.2 节将会进一步探讨 Netty 和 RxJava。

1.5 反应式抽象

实际上，RxJava 类型和操作符仅仅是对命令式回调的抽象。但是，这种抽象完全改变了代码风格，并且提供了非常强大的工具来实现异步和非阻塞编程。需要付出一定的努力才能掌握它，并且需要转换思维方式才能熟悉函数组合和流式思考。但是在掌握之后，它可以作为面向对象和命令式编程风格之外的一个非常有效的工具。

本书的其余部分将会介绍 RxJava 运行和使用的细节。第 2 章将阐述 `Observable` 是如何产生的，以及如何使用它们。第 3 章会讲解一些声明式且功能强大的转换。

注 1：当事件循环的数量是内核数量的 1 倍、1.5 倍或 2 倍时，这一点无可争辩。我至今未发现这些值的明显差异，通常默认为 1 倍。

第2章

Reactive Extensions

托马什·努尔凯维茨（Tomasz Nurkiewicz）

本章将介绍 Reactive Extensions 和 RxJava 的相关核心概念。通过这一章的学习，你将会非常熟悉 Observable<T>、Observer<T> 和 Subscriber<T>，并且学会一些实用的工具方法，即**操作符**（operator）。Observable 是 RxJava 的核心 API，所以一定要理解它是如何工作的，以及它代表了什么。在本章中，你将会学习 Observable 到底是什么，如何创建它，以及如何与之交互。这些知识对于按照习惯用法提供和使用基于 RxJava 的反应式 API 至关重要。设计 RxJava 是为了缓解异步和事件驱动编程的痛苦，但是为了更好地使用它，必须理解一些核心的原则和语义。掌握了 Observable 如何与客户端代码协作之后，你就会发觉自己的编码功力大有长进。阅读完本章，你将能够创建简单的数据流，并且能以非常有趣的方式对它们进行联结和组合。

2.1 剖析rx.Observable

rx.Observable<T> 代表了一个流形式的值序列，这一抽象形式我们会一直使用。因为这些值通常会在很长一段时间内出现，所以我们一般倾向于将 Observable 想象为一个事件流。如果你看看周围，就会发现很多流的例子。

- 用户界面事件。
- 网络上的字节传输。
- 在线商店的新订单。
- 社交网站的帖子。

如果想将 Observable<T> 与你更熟悉的事物进行类比，那么 Iterable<T> 可能是最接近的

抽象形式。就像 Iterable<T> 产生的 Iterator<T> 一样，Observable<T> 也可以有零到无穷个类型为 T 的值。Iterator 能够非常有效地生成无穷序列，例如，所有的自然数，如下所示。

```
class NaturalNumbersIterator implements Iterator<BigInteger> {

    private BigInteger current = BigInteger.ZERO;

    public boolean hasNext() {
        return true;
    }

    @Override
    public BigInteger next() {
        current = current.add(BigInteger.ONE);
        return current;
    }
}
```

另外一个相似之处就是 Iterator 自身可以给客户端发送信号，表明它没有更多的条目要生成了（稍后将会详细介绍）。但是，它们的相似性仅限于此。Observable 本质上是基于推送的，这意味着由它决定何时生成值。而 Iterator 则会一直处于空闲状态，直到有人调用 next() 条目。传统上来讲，Observable 不可能实现这种行为——在某个时间点，客户端代码可以**订阅**一个 Observable，当 Observable 感觉应该发布一个值的时候，订阅者就会得到通知。至于发布值的时间，可能是立即执行，也可能是永远不执行。6.2 节会讨论“回压”（backpressure）。借助这种机制，Subscriber 能够在特定的场景下控制 Observable 发布值的节奏。

同样，Observable 能够产生任意数量的事件。显然，这与经典的观察者（observer）模式非常类似，该模式也被称为发布 – 订阅模式。如果你想了解该模式的更多信息，请阅读 Erich Gamma 和 Richard Helm 合著的《设计模式：可复用面向对象软件的基础》。但是，就像 Iterator 不一定要由底层的集合作为支撑一样（参见 NaturalNumbersIterator），Observable 也不一定需要代表事件流。接下来看一些 Observable 的样例。

- Observable<Tweet> tweets

  tweets 可能是最明显的事件流的样例了。我们立刻明白，任何社交媒体网站的状态都在不停地更新，所以它们当然能够以事件流的形式来表示。不同于 Iterator，我们不能手动拉取需要的数据，Observable 必须在数据到达的时候将其推送给我们。

- Observable<Double> temperature

  temperature Observable 与 tweets 非常类似，它会生成某个设备的温度值并将其推送给订阅者。tweets 和 temperature Observable 都是由未来事件组成的无穷流的样例。

- Observable<Customer> customers

  Observable<Customer> 代表的内容取决于上下文。它最可能返回的是从数据库查询得到的一个客户列表，它可能是零个、多个甚至上千个可能延迟加载的项。这个 Observable 还可能代表要记录到系统中的 Customer 日志流。不管 Observable<Customer> 如何实现，客户端编程模型不会发生变化。

- ❑ `Observable<HttpResponse> response`

  与上面介绍的不同，`Observable<HttpResponse>` 很可能在终止的时候只生成一个事件（值）。这个值会在未来某个时间点出现，并且会被推送至客户端代码。要阅读这个响应，我们必须进行订阅。

- ❑ `Observable<Void> completionCallback`

  最后，是看上去有些诡异的 `Observable<Void>`。从技术上来讲，`Observable` 可以不发布任何条目就终止。在这种情况下，我们并不关心 `Observable` 推送的实际类型，因为它永远不会出现。

实际上，`Observable<T>` 可以生成三种类型的事件。

- `Observatble` 声明的类型为 `T` 的值。
- 完成事件。
- 错误事件。

Reactive Extensions 规范明确规定，所有 `Observable` 都可以发布任意数量的值，并且可以跟随一个完成或错误事件（但是不能两者兼有）。严格来说，Rx 设计指南将该规则定义成了如下的形式：`OnNext*(OnCompleted | OnError)?`。其中，`OnNext` 代表一个新的事件。有意思的是，对这个类似正则表达式的规则的任意可行的组合都是合法且有用的。

- ❑ `OnNext OnCompleted`

  `Observable` 发布一个值并优雅地终止。当 `Observable` 代表一个对外部系统的请求，并且我们预期得到单个响应时，可以采用这种形式。

- ❑ `onNext+ OnCompleted`

  `Observable` 在终止之前会发布多个值。它可以代表从数据库中读取一个列表，并且将每个条目作为单个值。另外一个例子是，如果某个进程要长期运行并最终能够完成，可以使用它来跟踪进程。

- ❑ `OnNext+`

  事件的无穷列表，比如社交媒体网站上的评论或某个组件的状态更新（如鼠标移动和 ping 请求）。这种流是无穷的，并且必须在运行时消费。

- ❑ 只有 `OnCompleted` 或 `OnError`

  这类 `Observable` 只表示正常或非正常的终止。`OnError` 还包含了导致流终止的 `Throwable`。错误是通过事件发出的，而不是标准的 `throw` 语句。

- ❑ `OnNext+ OnError`

  某个流可能会成功地发布一个或多个事件，但最终会失败。一般而言，这意味着这个流应该是无穷的，但期间因为某种致命错误出现了故障。可以想象成一组网络数据包，它要持续投递事件数个小时，但是在某个时间点它可能因为连接丢失而中断。

`OnError` 通知非常有意思。因为 `Observable` 异步的特性，简单地抛出异常并没有太大的意义。相反，必须要将错误传输给对其感兴趣的人，这可能会跨线程并且需要一段时间。`OnError` 是特殊类型的事件，它以函数式的方式封装了异常。7.1 节会介绍关于异常的更多内容。

除此之外，你还可以实现不发布任何事件的 Observable，包括完成或错误事件。这样的 Observable 适用于测试，如练习超时。

2.2 订阅来自Observable的通知

在其他人对 Observable 发出的事件表示感兴趣之前，Observable 实例不会发布任何事件。要开始观察一个 Observable，可以使用 subscribe() 方法族。

```
Observable<Tweet> tweets = //...

tweets.subscribe((Tweet tweet) ->
    System.out.println(tweet));
```

上面的代码片段通过注册一个回调来订阅 tweets Observable。每次 tweets 流决定推送一个事件到下游，这个回调就会被调用。RxJava 契约会确保你的回调不会同时在多个线程中触发，即便事件是从多个线程中发布的。subscribe() 有多个重载版本，它们更加具体。前文已经提到过，Observable 一般不会抛出异常。相反，异常是 Observable 能够传播的另一种通知（事件）类型。所以，你不能围绕着 subscribe() 使用 try-catch 代码块来捕获这个过程中的异常，而是提供一个单独的回调，如下所示。

```
tweets.subscribe(
        (Tweet tweet) -> { System.out.println(tweet); },
        (Throwable t) -> { t.printStackTrace(); }
);
```

subscribe() 的第二个参数是可选的。它会通知在生成条目的时候可能会抛出的异常，保证在这个异常之后，不会有其他的 Tweet 出现。即便你预计异常不会出现，也应该始终订阅异常，而不仅仅订阅合法的条目。在 Observable 中，异常是一等公民。抛出的异常可以快速传播，产生很多的副作用，比如不一致的数据结构或失败的事务。一般来说，这是很好的方法，但异常通常并不是致命的。因此，可靠的系统应该预先谋划并采用系统性的方式来处理异常。这就是 Observable 将其显式建模的原因。

第三个可选的回调让我们能够监听流的结束。

```
tweets.subscribe(
        (Tweet tweet) -> { System.out.println(tweet); },
        (Throwable t) -> { t.printStackTrace(); },
        () -> {this.noMore();}
);
```

需要记住，RxJava 对于生成多少条目、何时生成以及何时停止并没有预设的限制。流可能是无穷的，也可能一旦订阅就马上结束，这取决于 Subscriber 是否想接收完成通知。如果你一开始就知道某个流是无穷的，那么订阅完成通知就没有意义了。另一方面，在某些场景中，流结束可能恰好是实际要等待的事件。举例来说，想想跟踪长期运行的进程的 Observable <Progress>。无论客户端对于跟踪进程是否感兴趣，它肯定想要知道进程何时结束。

补充一句，通常可以使用 Java 8 的方法引用代替 lambda 表达式，以提升代码的可读性，如下所示。

```
tweets.subscribe(
    System.out::println,
    Throwable::printStackTrace,
    this::noMore);
```

使用Observer<T>捕获所有的通知

实践证明，为 subscribe() 提供所有三个参数是非常有用的。因此，含有这三个回调的包装器也会非常有用。这就是设计 Observer<T> 的目的。Observer<T> 是这三个回调的容器，能够接收来自 Observable<T> 的所有可能的通知。如下的代码展现了如何注册 Observable<T>。

```
Observer<Tweet> observer = new Observer<Tweet>() {
    @Override
    public void onNext(Tweet tweet) {
        System.out.println(tweet);
    }

    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }

    @Override
    public void onCompleted() {
        noMore();
    }
};

//...

tweets.subscribe(observer);
```

实际上，Observer<T> 是 RxJava 监听功能的核心抽象。不过，如果你想要更好地进行控制，Subscriber（Observer 抽象的实现）更加强大。

2.3 使用Subscription和Subscriber<T>控制监听器

一个 Observable 可以有多个订阅者。类似于发布者 – 订阅者模式，一个发布者可以分发事件给多个消费者。在 RxJava 中，Observable<T> 只是一个类型化数据结构，它可能存活很短的时间，也可能只要服务器程序在运行，它就持续存活。对于订阅者来说也是如此。你可以订阅一个 Observable，消费一些事件，抛弃其他的事件。也可以采用完全相反的做法：只要 Observable 存活，就一直抽取事件，或许能持续几个小时或几天。

假设某个 Observer 预先知道它想要接收多少条目或者何时停止接收。比如，我们订阅了股票价格变化的消息，但是如果价格低于 1 美元，我们将不再监听。显然，Observer 有订阅的能力，那么它也应该具有在合适的情况下取消订阅的能力。有两种方式支持该功能：Subscription 和 Subscriber。先讨论一下前者。这里不探讨 subscribe() 实际会返回什么。

```
Subscription subscription =
        tweets.subscribe(System.out::println);

//...

subscription.unsubscribe();
```

Subscription 是一个句柄，允许客户端代码通过 unsubscribe() 方法取消订阅。除此之外，你还可以通过 isUnsubscribed() 查询订阅的状态。不想接收更多事件的时候，取消对 Observable<T> 的订阅是非常重要的，这能够避免内存泄漏和不必要的系统负载。有时候我们会订阅并完整消费一个 Observable，即便这个流是无穷的，也永远不会真正取消订阅。但是，也存在订阅者来来去去，Observable 却在持续生成事件的情况。

还有第二种方式能取消订阅，这次是在监听者内部实现。我们已经知道，可以通过 Subscription 在 Observer 或回调之外控制订阅情况。而 Subscriber<T> 同时实现了 Observer<T> 和 Subscription。因此，它既能够用来消费通知（事件、完成或错误信息），又能够控制订阅。如下的示例代码订阅了所有的事件，但是订阅者本身会在满足特定标准的时候放弃接收通知。正常情况下，可以通过内置的 takeUntil() 操作符来完成该功能，目前也可以手动取消订阅。

```
Subscriber<Tweet> subscriber = new Subscriber<Tweet>() {
    @Override
    public void onNext(Tweet tweet) {
        if (tweet.getText().contains("Java")) {
            unsubscribe();
        }
    }

    @Override
    public void onCompleted() {}

    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }
};
tweets.subscribe(subscriber);
```

Subscriber 决定不再接收更多条目时，它可以自己取消订阅。作为练习，你可以实现一个 Subscriber，它只接收前 n 个事件，然后放弃接收事件。Subscriber 类还有更强大的功能，但是目前记住它能够自己取消对 Observable 订阅即可。

2.4 创建Observable

本书首先介绍了订阅 Observable 以接收那些推送到下游的事件。这并非巧合，在使用 RxJava 的大多数情况下，我们都会与已有的 Observable 交互，一般会对它们进行组合、过滤和包装。除非你使用的外部 API 暴露了 Observable，否则你必须先了解 Observable 来自哪里，以及如何创建流和处理订阅。首先，有一些工厂方法能够创建固定的常量 Observable。如果你想在整个代码库中前后一致地使用 RxJava，或者要发布的值生成成本

很低并预先知道它的内容，那么这种方式是很有用的。

- Observable.just(value)
  创建一个 Observable 实例，它只给所有未来的订阅者发布一个值，随后正常完成。重载版本的 just() 操作符还能发布 2 到 9 个值。
- Observable.from(values)
  类似于 just()，但是它接受 Iterable<T> 或 T[] 作为参数，因此它创建的 Observable<T> 发布的值就是 values 集合中的元素。另外一个重载版本接受 Future<T>，底层 Future 完成的时候会发布一个事件。
- Observable.range(from, n)
  从 from 开始生成 n 个整型数字。例如，range(5, 3) 将会发布 5、6 和 7，然后正常完成。每个订阅者会接收到一组相同的数字。
- Observable.empty()
  订阅后立即完成，不发布任何的值。
- Observable.never()
  这样的 Observable 不发布任何的通知，无论是值，还是完成或失败。这个流适用于测试。
- Observable.error()
  立即给每个订阅者发送一个 onError() 通知。不发布任何的值，按照契约，也不会发送 onCompleted() 通知。

2.4.1 掌握Observable.create()

empty()、never() 和 error() 工厂方法看上去似乎没有太大的用处，但是它们与真正的 Observable 组合时是非常便利的。有意思的是，尽管 RxJava 就是异步处理事件流，上述的工厂方法却默认是在客户端线程中执行的。请看下述的代码样例。

```
private static void log(Object msg) {
    System.out.println(
          Thread.currentThread().getName() +
          ": " + msg);
}

//...

log("Before");
Observable
    .range(5, 3)
    .subscribe(i -> {
        log(i);
    });
log("After");
```

我们感兴趣的是执行每个日志语句的线程。

```
main: Before
main: 5
```

```
main: 6
main: 7
main: After
```

print 语句的顺序也是值得关注的。Before 和 After 消息是由 main 客户端线程打印出来的，这一点倒不令人惊讶。但是，请注意订阅也是发生在客户端线程中的，subscribe() 实际上会阻塞客户端线程，直到所有的事件都被接收。除非某些操作符需要，否则 RxJava 不会隐式地在线程池中运行代码。为了更好地理解这种行为，接下来学习用来生成 Observable 的一个低层级的操作符，即 create()。

```
Observable<Integer> ints = Observable
    .create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            log("Create");
            subscriber.onNext(5);
            subscriber.onNext(6);
            subscriber.onNext(7);
            subscriber.onCompleted();
            log("Completed");
        }
    });

log("Starting");
ints.subscribe(i -> log("Element: " + i));
log("Exit");
```

上面的示例代码有意写得比较冗长。如下是输出，包括执行每行代码的线程名称。

```
main: Starting
main: Create
main: Element: 5
main: Element: 6
main: Element: 7
main: Completed
main: Exit
```

为了理解 Observable.create() 是如何运行的，以及 RxJava 是如何处理并发的，接下来将逐行分析它的执行过程。首先，样例为 create() 方法提供一个 OnSubscribe 回调接口的实现，以创建名为 ints 的 Observable（稍后，基本上会用简单的 lambda 表达式来替代它）。现在，除了创建 Observable 实例之外，还没有发生任何事情，所以第一行输出的是 main: Starting。Observable 默认会延迟事件的发布，所以为 create() 提供的 lambda 表达式并没有执行。随后，订阅 ints.subscribe(...)，强制 Observable 开始发布条目。对于 cold 类型的流来说，这些情况都是成立的。但是，对于 hot 类型的流来说，情况会有所不同：不管有没有人订阅，它都会发布事件。2.4.4 节将会阐述它们的差别。

接收发布条目的 lambda 表达式 (i -> log("Element: " + i) 在内部会被 Subscriber<Integer> 包装。调用 create() 时，这个订阅者基本上会直接作为参数传递给指定的函数。所以，每订阅一个 Observable，就会创建一个新的 Subscriber 实例并传递给 create() 方法。在 create() 中，调用 onNext() 或 Subscriber 的其他方法会间接触发你自己的 Subscriber。

Observable.create() 的用途非常广泛，实际上，你可以基于它模拟之前提到的各个工厂方法。例如，Observable.just(x) 会发布单个值 x 并立即结束，看起来可能会是这样的：

```
static <T> Observable<T> just(T x) {
    return Observable.create(subscriber -> {
            subscriber.onNext(x);
            subscriber.onCompleted();
        }
    );
}
```

作为练习，请尝试实现 never()、empty()，甚至仅用 create() 实现 range()。

管理多个订阅者

如果没有实际订阅，事件是不会发布的。但是，每次调用 subscribe() 的时候，create() 中的订阅句柄也会被调用。这说不上是什么优势或劣势，只是需要记住这一点。在有些场景下，每个订阅者都能对应唯一的句柄调用是非常棒的。例如，Observable.just(42) 会将 42 发布给每个订阅者，而不是只给第一个订阅者。另一方面，如果你要在 create() 中进行数据库查询或重量级的计算，那么所有的订阅者共享同一次调用就会带来一定的收益。

为了保证你真正理解订阅是如何运行的，不妨考虑如下的样例，它对同一个 Observable 订阅了两次。

```
Observable<Integer> ints =
        Observable.create(subscriber -> {
                    log("Create");
                    subscriber.onNext(42);
                    subscriber.onCompleted();
                }
        );
log("Starting");
ints.subscribe(i -> log("Element A: " + i));
ints.subscribe(i -> log("Element B: " + i));
log("Exit");
```

你预期的输出是什么样子的呢？记住，每当订阅通过 create() 工厂方法生成的 Observable，作为参数传递给 create() 的 lambda 表达式都会默认在初始化订阅的线程中独立执行。

```
main: Starting
main: Create
main: Element A: 42
main: Create
main: Element B: 42
main: Exit
```

如果你不想为所有订阅者调用 create()，而是想重用已经计算的事件，那么可以使用非常便利的 cache() 操作符。

```
Observable<Integer> ints =
    Observable.<Integer>create(subscriber -> {
                //...
            }
    )
    .cache();
```

cache() 是我们学习的第一个操作符。操作符会包装并增强已有的 Observable，一般通过对订阅进行拦截来实现。cache() 做的事情位于 subscribe() 和自定义的 Observable 之间。第一个订阅者出现时，cache() 将订阅委托给底层的 Observable，然后将所有的通知（事件、完成或错误）转发给下游。但是，它同时还会在内部保留所有通知的一个副本。如果后续的订阅者希望接收已推送的通知，cache() 不会再委托给底层的 Observable，而是将缓存的值发布出来。在使用缓存时，两个 Subscriber 的输出会有很大的差异，如下所示。

```
main: Starting
main: Create
main: Element A: 42
main: Element B: 42
main: Exit
```

当然，一定要记住 cache() 和无穷流组合将会带来灾难性的结果，也就是 OutOfMemoryError。8.6 节将会对其进行介绍。

2.4.2 无穷流

无穷的数据结构是很重要的概念。计算机的内存是有限的，因此无穷的列表或流听起来似乎是不可能实现的。但是，RxJava 允许动态地生成和消费事件。传统队列可以视为无穷值的来源，只不过它不会同时将所有的值都保存在内存之中。那么，该如何使用 create() 实现这样的无穷流呢？例如，创建一个生成所有自然数的 Observable。

```
//有问题！不要这样做
Observable<BigInteger> naturalNumbers = Observable.create(
        subscriber -> {
            BigInteger i = ZERO;
            while (true) { //不要这样做!
                subscriber.onNext(i);
                i = i.add(ONE);
            }
        });
naturalNumbers.subscribe(x -> log(x));
```

在任何的代码库中，while(true) 这种编码方式都会引发告警。这看上去没有毛病，但很快你就会意识到这个实现是有问题的。不过，这并非因为它是无穷的，实际上无穷的 Observable 是完全可以的，而且非常有用。当然，前提是它要以恰当的方式来实现。调用 subscribe() 时，create() 中的 lambda 表达式将会在你的线程的上下文中执行。因为这个 lambda 表达式永远不会结束，subscribe() 也就会永远地阻塞。你可能会问：“订阅难道不应该是异步的吗？那为什么会在客户端线程中运行订阅句柄呢？”这是一个很合理的问题，所以下面花点时间来介绍显式的并发，如下所示。

```
Observable<BigInteger> naturalNumbers = Observable.create(
    subscriber -> {
        Runnable r = () -> {
            BigInteger i = ZERO;
            while (!subscriber.isUnsubscribed()) {
                subscriber.onNext(i);
                i = i.add(ONE);
            }
```

```
        };
        new Thread(r).start();
    });
```

这里不再是在客户端线程中直接运行阻塞循环，而是生成了一个自定义的线程，在该线程中发布事件。现在，subscribe() 不会再阻塞客户端线程，因为它底层做的事情仅仅是生成一个线程。x -> log(x) 回调的所有调用都会在自定义线程的后台执行。假设现在我们不再对所有的自然数感兴趣（毕竟它们的数量太多了），只对前面的一部分感兴趣。我们已经知道了如何在 Observable 中停止接收通知——通过取消订阅功能。

```
Subscription subscription = naturalNumbers.subscribe(x -> log(x));
//一段时间之后……
subscription.unsubscribe();
```

如果你关注细节，会发现看上去非常可疑的 while(true) 循环已经被替换成以下代码。

```
while (!subscriber.isUnsubscribed()) {
```

我们发起的每一次迭代，都需要确保有人在进行监听。某个订阅者决定停止监听时，subscriber.isUnsubscribed() 条件就会发出通知，这样，我们就能安全地完成流并退出 Runnable，实际上也就是停止了该线程。显然，每个订阅者都有自己的线程和循环，所以即使某个订阅者决定取消订阅，其他的订阅者依然能够接收其独立的事件流。尽管创建自己的线程并不是好的设计决策，RxJava 也有更好的声明式工具来处理并发，但是上述的代码样例依然展现了如何恰当地处理订阅事件。

建议尽可能频繁地检查 isUnsubscribed() 标记，从而避免将事件发送给那些已经不再想接收新事件的订阅者。此外，当生成事件的成本很高的时候，如果没有人想接收事件，再急切地将它们发送出来就没有意义了。自行生成线程的做法本质上没有什么问题，但是它容易出错，并且扩展性很差。4.9 节会探讨声明式并发和自定义调度器。这些特性允许我们编写并发的代码，而不必自己与线程进行真正的交互。

如果事件推送的频率相对很高，在发送事件之前处理取消订阅是一种不错的办法。但是，假设在一个场景中事件发生的频率非常低，Observable 只有在试图推送事件的时候才能判断订阅者是否已经取消了订阅。以下面这个很有用的工厂方法为例：delayed(x) 会创建一个 Observable，该 Observable 在休眠 10 秒之后会发布一个值 x。它类似于 Observable.just()，但有额外的延迟。这时需要一个额外的线程，尽管这并不是最佳的使用模式，如下所示。

```
static <T> Observable<T> delayed(T x) {
    return Observable.create(
        subscriber -> {
            Runnable r = () -> {
                sleep(10, SECONDS);
                if (!subscriber.isUnsubscribed()) {
                    subscriber.onNext(x);
                    subscriber.onCompleted();
                }
            };
            new Thread(r).start();
        });
```

```
}

static void sleep(int timeout, TimeUnit unit) {
    try {
        unit.sleep(timeout);
    } catch (InterruptedException ignored) {
        //有意忽略
    }
}
```

这里的原始实现会生成一个新的线程并休眠 10 秒。更为健壮的实现方式至少要使用 `java.util.concurrent.ScheduledExecutorService`，但仅用于教学。在 10 秒之后，确保依然还有人在监听，如果确实如此，就发布一个条目并完成。但是，如果订阅者在订阅 1 秒之后就决定取消订阅会怎么样呢，也就是比理应发布事件的时间早得多？这也没什么。后台线程继续休眠 9 秒，然后才会意识到订阅者已经不存在了。这里就是令人困扰的地方，额外再持有资源 9 秒是非常浪费的。假设这是一个到数据 feed 的连接，它的成本非常高，使用它需按秒付费，但是事件的发生频率却非常低。等待数秒，甚至数分钟，才能发现没有人订阅并终止连接。这样的做法并不是最优的。

幸而，借助 `subscriber` 实例可以在取消订阅之时马上得到通知，并立即清理资源，不用等到下一条消息出现才进行这些操作。

```
static <T> Observable<T> delayed(T x) {
    return Observable.create(
        subscriber -> {
            Runnable r = () -> {/* ... */};
            final Thread thread = new Thread(r);
            thread.start();
            subscriber.add(Subscriptions.create(thread::interrupt));
        });
}
```

最后一行最重要，但其他的内容也不能忽略。后台线程已经运行了，或者更精确地说，它已经开始 10 秒休眠了。但是，生成线程之后，通过一个回调要求订阅者告知我们它是否取消了订阅，这是通过 `Subscriber.add()` 注册的。这个回调只有一个很简单的目的：中断线程。调用 `Thread.interrupt()` 之后，将会在 `sleep()` 中抛出 `InterruptedException`，提前中断 10 秒休眠。`sleep()` 在接收到这个异常之后，就会优雅地退出。但是，此时 `subscriber.isUnsubscribed()` 返回 `true`，并没有发出任何事件。线程马上结束，不会浪费任何资源。你可以采用这种模式执行任意的清理工作。但是，如果流生成的是稳定、频繁的事件，那么你可能就不需要显式回调。

还有一个不应该在 `create()` 中使用显式线程的原因。Rx 设计指南的 4.2 节（“假定观察者实例以序列化的方式被调用”）要求订阅者不能并发地接收通知。涉及显式的线程时，很容易违反这个要求。这种行为类似于 Akka 工具集中的 `Actor`，在那里每个 `Actor` 一次只能处理一条消息。按照这样的假设，可以编写同步的 `Observer`，通常只能由一个线程进行访问。即便事件是由多个线程发出的，这个假设依然成立。`Observable` 的自定义实现必须要确保满足这个契约。如下的代码试图并行加载多个块的 `Data`，这并不符合习惯的用法。

```
Observable<Data> loadAll(Collection<Integer> ids) {
    return Observable.create(subscriber -> {
        ExecutorService pool = Executors.newFixedThreadPool(10);
        AtomicInteger countDown = new AtomicInteger(ids.size());
        //危险，这违反了Rx契约。不要这样做！
        ids.forEach(id -> pool.submit(() -> {
            final Data data = load(id);
            subscriber.onNext(data);
            if (countDown.decrementAndGet() == 0) {
                pool.shutdownNow();
                subscriber.onCompleted();
            }
        }));
    });
}
```

这段代码除了特别复杂之外，还违反了一些 Rx 原则。也就是说，它允许从多个线程并发地调用 `subscriber` 的 `onNext()` 方法。其次，使用 RxJava 的惯用操作符就可以避免这种复杂性，比如 `merge()` 和 `flatMap()`，但是 3.2.1 节才会对其进行讨论。好消息是，即便有人将 `Observable` 实现得非常糟糕，我们依然可以通过 `serialize()` 操作符来修正它，比如 `loadAll(...).serialize()`。这个操作符能够确保事件是序列化和有序的，它还能确保完成和错误之后，不会再有事件发出。

在创建 `Observable` 方面，还没有提及的一个问题是错误传播。到目前为止，本章已经介绍了 `Observer<T>` 能够接收 `T` 类型的值，还有可选的完成或错误事件。但是，该如何将错误推送至下游所有的订阅者呢？最佳的做法是在 `create()` 方法中，将所有的表达式都包装在一个 `try-catch` 代码块中。`Throwable` 应该传播至下游，而不是打印日志或重新抛出，如下所示。

```
Observable<Data> rxLoad(int id) {
    return Observable.create(subscriber -> {
        try {
            subscriber.onNext(load(id));
            subscriber.onCompleted();
        } catch (Exception e) {
            subscriber.onError(e);
        }
    });
}
```

额外的 `try-catch` 代码块是非常必要的，它会传播可能被抛出的 `Exception`，比如 `load(id)` 抛出的 `Exception`。否则，RxJava 最多只能将异常打印在标准输出上。但是为了构建弹性流，异常需要被当作一等公民对待，而不能仅仅是编程语言中没有人能真正理解的额外特性。

`Observable` 通过一个值来结束，并且使用 `try-catch` 来进行包装是非常常见的模式，所以 RxJava 引入了内置的 `fromCallable()` 操作符。

```
Observable<Data> rxLoad(int id) {
    return Observable.fromCallable(() ->
        load(id));
}
```

在语义上，它与前面的代码相同，但是更加简短。除此之外，相对于 create()，它还有一些其他的优点，稍后会介绍。

2.4.3 计时：timer()和interval()

我们已经介绍了 Observable 如何自行创建线程，但它在 RxJava 中并不是最好的模式。后面的章节会介绍调度器，但是我们先来了解一下两个非常有用的操作符，也就是 timer() 和 interval()，它们在底层会用到线程。前者只是简单地创建一个 Observable，在特定的延迟后发布一个 long 类型的零值，然后完成。

```
Observable
        .timer(1, TimeUnit.SECONDS)
        .subscribe((Long zero) -> log(zero));
```

虽然看起来很傻，但是 timer() 非常有用。基本上，它就是一个异步版本的 Thread.sleep()。我们创建了一个 Observable 并通过 subscribe() 订阅它，而不是阻塞当前线程。学习如何将简单的 Observable 组合为更复杂的计算形式之后，它将变得更加重要。固定的值零（在 zero 变量中）只是一个约定，并没有任何特殊的含义。不过，引入 interval() 之后，就会更有意义了。interval() 会生成一个 long 类型数字的序列，从零开始，每个数字之间有固定的时间间隔。

```
Observable
        .interval(1_000_000 / 60, MICROSECONDS)
        .subscribe((Long i) -> log(i));
```

Observable.interval() 会从零开始生成一系列 long 类型的连续数字。但是，与 range() 不同，interval() 会在每个事件之前添加固定的时间间隔，包括第一个。样例中的这个延迟大约是 16 666 μs，大致对应为 60 Hz，这也是各种动画中常用的帧率。这并不是巧合：interval() 有时用来控制需要以特定频率运行的动画或进程。某种程度上，interval() 类似于 ScheduledExecutorService 中的 scheduleAtFixedRate()。你可以想象一下 interval() 的多种使用场景，比如定期轮询数据、刷新用户界面或者建模时间的推移。

2.4.4 hot和cold类型的Observable

在得到一个 Observable 实例之后，很重要的一点是要理解它是 hot 类型的还是 cold 类型的。它们的 API 和语义是相同的，但是使用 Observable 的方法取决于它的类型。cold 类型的 Observable 完全是延迟（lazy）执行的，并且在有人对其感兴趣时才会开始发布事件。如果没有观察者，那么 Observable 只是一个静态的数据结构。这也意味着，每个订阅者都会接收到属于自己的流的副本，因为事件是延迟生成的，并且一般不会采取任何形式的缓存。cold 类型的 Observable 通常来源于 Observable.create()，按照语义，它不会启用任何逻辑，而是推迟到有人实际对其监听才会执行。从某种程度上来说，cold 类型的 Observable 依赖 Subscriber。cold 类型的 Observable 的样例除了 create() 之外，还包括 Observable.just()、from() 和 range()。订阅一个 cold 类型的 Observable 通常还涉及 create() 中的副作用，比如查询数据库或打开连接。

hot 类型的 Observable 则与之不同。在得到这种类型的 Observable 的时候，不管是否

有 Subscriber，它都可能已经开始发布事件了。即便没有人监听，事件可能会丢失，Observable 依然会往下游推送事件。通常情况下，可以完全控制 cold 类型的 Observable，但是 hot 类型的 Observable 是独立于消费者的。Subscriber 出现时，hot 类型的 Observable 的行为类似于电话窃听（wire tap），[1] 透明地发布流经它的事件。Subscriber 的出现和消失并不会改变 Observable 的行为，它是完全解耦和独立的。

稍感意外的是，Observable.interval() 并不是 hot 类型的 Observable。你可能认为它只是每隔固定的时间生成报时信号，与环境并没有什么关系，但实际上，只有有人订阅的时候，才会生成计时器事件，而每个订阅者会得到独立的流。这完全符合 cold 类型的 Observable 的定义。

hot 类型的 Observable 通常发生在完全无法控制事件源的场景下。这种 Observable 的样例包括鼠标移动、键盘输入或按钮点击。到目前为止，还没有提及用户界面，但事实上，RxJava 非常适合用户界面的实现。这个库在 Android 社区特别受欢迎，它能够帮助开发人员将嵌套回调转换为扁平化的流组合。8.1 节将探索如何在运行 Android 系统的移动设备上使用 RxJava。

依赖事件传递时，hot 类型和 cold 类型 Observable 的差异就变得非常重要了。不管立即订阅还是几个小时之后订阅 cold 类型的 Observable，你都会获得完整且一致的事件集。但如果 Observable 是 hot 类型的，那么你就无法确保能接收到所有事件。本章稍后将介绍一些技术，它们能够确保每个订阅者都能接收到所有事件，其中一项本章已经介绍过了：cache() 操作符（参见 2.4.1 节）。在技术上来讲，可以缓冲来自 hot 类型 Observable 的所有事件，让后续的订阅者都能接收到相同的事件序列。但是，在理论上，它消耗的内存量是没有限制的，所以在缓存 hot 类型的 Observable 时要非常小心。

hot 源和 cold 源的时间依赖性是另外一个值得注意的差异。cold 类型的 Observable 按需生成值，并且可能会生成多次，所以某个条目的确切生成时间无关紧要。但是，hot 类型的 Observable 一般代表的是外部源生成的事件。这意味着给定值的生成时间是非常重要的，因为它能够将事件限定在一个时间范围。

2.5　用例：从回调API到Observable流

大多数的 Java API 是阻塞式的，比如 JDBC、java.io、Servlet[2]，以及私有的解决方案。这意味着，不管结果或副作用是什么，客户端线程必须要等待。但是，有些用例本质上就是异步的，比如推送来自外部源的事件。从技术上来讲，你可以按照如下的方式构建阻塞式的流 API。

```
while(true) {
    Event event = blockWaitingForNewEvent();
    doSomethingWith(event);
}
```

如果某个领域本质上是异步的，如 JavaScript，你很可能会发现某种基于回调的 API 非常

注 1：Gregor Hohpe 和 Bobby Woolf 编著的《企业集成模式：设计、构建及部署消息传递解决方案》。

注 2：至少直到 3.0 版本。

常见。这些 API 会接收某种形式的回调，一般而言是带有一组方法的接口，从而实现各种事件的通知。几乎所有的用户界面都是这种 API 的典型例子，比如 Swing。在使用像 onClick() 或 onKeyUp() 这样的监听器之后，回调就难以避免了。如果你曾经经历过这样的环境，那么你对**回调地狱**（callback hell）这个术语肯定不会感到陌生。回调很容易互相嵌套，所以协调多个回调几乎是不可能的。如下的样例展现了回调嵌套 4 层的场景。

```
button.setOnClickListener(view -> {
    MyApi.asyncRequest(response -> {
        Thread thread = new Thread(() -> {
            int year = datePicker.getYear();
            runOnUiThread(() -> {
                button.setEnabled(false);
                button.setText("" + year);
            });
        });
        thread.setDaemon(true);
        thread.start();
    });
});
```

在以上的代码中，即便是最简单的需求都将是一场噩梦，比如要在两个回调依次调用之后做出反应，再加上多线程的阻碍就更是雪上加霜了。本节会将基于回调的 API 重构为 RxJava，它具备所有优点，比如线程控制、生命周期管理和清理。

关于流，我最喜欢的一个例子就是 Twitter 的状态更新，也就是所谓的推文（tweet）。每秒都会有数千个用户更新，很多更新会伴随着地理位置、语言和其他元数据。为了完成这个练习，将会使用开源的 Twitter4J 库，它使用基于回调的 API 将新推文的子集推送过来。本章的目的并不是阐述 Twitter4J 是如何运行的，也不会提供健壮的例子。选择 Twitter4J 是因为它是使用回调 API 的一个很好的范例，并且它的领域非常有意思。实时读取推文的最简单的可运行样例如下所示。

```
import twitter4j.Status;
import twitter4j.StatusDeletionNotice;
import twitter4j.StatusListener;
import twitter4j.TwitterStream;
import twitter4j.TwitterStreamFactory;

TwitterStream twitterStream = new TwitterStreamFactory().getInstance();
twitterStream.addListener(new twitter4j.StatusListener() {
    @Override
    public void onStatus(Status status) {
        log.info("Status: {}", status);
    }

    @Override
    public void onException(Exception ex) {
        log.error("Error callback", ex);
    }

    //其他回调
});
```

```
twitterStream.sample();
TimeUnit.SECONDS.sleep(10);
twitterStream.shutdown();
```

调用 twitterStream.sample() 将会启动一个后台线程，该线程会登录 Twitter 并等待新的消息。每次有推文出现，onStatus 回调就会执行。执行过程可能会跨线程，所以不能依赖异常抛出的机制，而是使用 onException() 通知。在休眠 10 秒之后，通过 shutdown() 关闭流并清理底层的资源，比如 HTTP 连接或线程。

整体而言，它看上去并没有那么糟糕，这个程序的问题在于什么都不做。在现实生活中，你可能会以某种方式处理每条 Status 消息（推文），比如保存到数据库中或者提供给一个机器学习算法。从技术上来讲，你可以将这些逻辑放到回调中，但是这样就将基础设施调用和业务逻辑耦合在一起了。简单地将功能委托给一个单独的类会更好一些，但很遗憾的是无法重用。我们真正想要的是技术领域（消费 HTTP 连接中的数据）和业务领域（解释输入的数据）的清晰分离。所以，我们构建了第二层回调。

```
void consume(
            Consumer<Status> onStatus,
            Consumer<Exception> onException) {
    TwitterStream twitterStream = new TwitterStreamFactory().getInstance();
    twitterStream.addListener(new StatusListener() {
        @Override
        public void onStatus(Status status) {
            onStatus.accept(status);
        }

        @Override
        public void onException(Exception ex) {
            onException.accept(ex);
        }

        //其他回调
    });
    twitterStream.sample();
}
```

通过添加这个额外的抽象层，现在能够以各种方式重用 consume() 方法。假设不再是进行日志记录，而是要进行持久化、分析或欺诈检测。

```
consume(
        status -> log.info("Status: {}", status),
        ex     -> log.error("Error callback", ex)
);
```

但是这只将问题在层级结构中进行了提升。如果想要记录每秒推文的数量该怎么办？或者只想消费前 5 条数据，又该怎样实现？如果想要有多个监听器，又会发生什么情况？前述每种情况都会打开一个新的 HTTP 连接。最后不得不提的是，API 不允许完成后再取消订阅，以免带来资源泄漏的风险。希望你能意识到，我们正在努力朝着基于 Rx 的 API 的方向努力。此时，不再传递回调到可能要执行的地方，而是返回一个 Observable<Status> 并允许每人按需对其进行订阅。但是，需要记住的一点是，如下的实现还是会为每个 Subscriber 打开一个新的网络连接。

```
Observable<Status> observe() {
    return Observable.create(subscriber -> {
        TwitterStream twitterStream =
            new TwitterStreamFactory().getInstance();
        twitterStream.addListener(new StatusListener() {
            @Override
            public void onStatus(Status status) {
                subscriber.onNext(status);
            }

            @Override
            public void onException(Exception ex) {
                subscriber.onError(ex);
            }

            //其他回调
        });
        subscriber.add(Subscriptions.create(twitterStream::shutdown));
    });
}
```

此时，只需调用 observe()，它只会创建一个 Observable，并且不会与外部的服务器联系。create() 的内容并不会执行，除非有人实际订阅。订阅也十分类似，如下所示。

```
observe().subscribe(
        status -> log.info("Status: {}", status),
        ex -> log.error("Error callback", ex)
);
```

以上代码与 consume(...) 的巨大差别在于，不必将回调作为参数传递给 observe()。相反，样例可以返回 Observable<Status>，将它到处传递，然后在某个地方进行存储，并且只要需要就可以随时随地使用。还可以将这个 Observable 与其他的 Observable 进行组合，这是第 3 章将会讲解的内容。本章还未讨论的一个方面是资源清理。有人取消订阅时，应当关闭 TwitterStream，以避免资源泄漏。我们已经知道了两种清理技术，下面首先使用稍微简单的一种。

```
@Override
public void onStatus(Status status) {
    if (subscriber.isUnsubscribed()) {
        twitterStream.shutdown();
    } else {
        subscriber.onNext(status);
    }
}

@Override
public void onException(Exception ex) {
    if (subscriber.isUnsubscribed()) {
        twitterStream.shutdown();
    } else {
        subscriber.onError(ex);
    }
}
```

如果有人进行订阅却只想得到流的一小部分，Observable 会确保进行资源清理。我们知道还有第二项实现清理的技术，它不需要等待上游的事件。订阅者取消订阅，样例就立即调用 shutdown() 以触发清理行为（最后一行），而不必等待下一条推文到达。

```
twitterStream.addListener(new StatusListener() {
    //回调……
});
twitterStream.sample();

subscriber.add(Subscriptions.create(twitterStream::shutdown));
```

比较有意思的是，这个 Observable 模糊了 hot 流和 cold 流的差异。一方面，它代表了外部的事件，这些事件的出现并不受控制（hot 流的行为）。另一方面，实际订阅之前，事件并不会开始流动（没有底层的 HTTP 连接）。我们还忽略了另外一个不慎出现的副作用：每个新的 subscribe() 都会开启一个新的后台线程和一条到外部系统的新连接。相同的 Observable<Status> 实例应该能够跨多个订阅者实现重用。因为 Observable 是延迟执行的，所以需要在技术上能够在启动的时候就调用 observe()，并将其保留在某个单例对象中。但是当前的实现只是简单地打开新连接，为每个 Subscriber 通过网络多次获取相同的数据。当然，我们想要为这个流注册多个 Subscriber，但是没有理由为每个 Subscriber 都独立地获取相同的数据。我们真正需要的是一种**发布－订阅**（pub-sub）行为，在这里一个发布者（外部系统）投递数据给多个 Subscriber。从理论上讲，cache() 操作符能够实现这一点，但是我们不希望永远缓存旧的事件。下面探讨一下这个问题的一些解决方案。

手动管理订阅者

手动跟踪所有的订阅者，并且在所有的订阅者都离开时关闭对外部系统的连接，这是一项没完没了的任务，我们尝试这种实现方式，仅仅是为了更好地阐述后文的惯用方案。以下代码的理念就是以 Set<Subscriber<Status>> 的某种形式跟踪所有的订阅者，并在该集合为空 / 非空的时候关闭 / 打开对外部系统的连接。

```
//不要这样做，很容易出错
class LazyTwitterObservable {

  private final Set<Subscriber<? super Status>> subscribers =
    new CopyOnWriteArraySet<>();

  private final TwitterStream twitterStream;

  public LazyTwitterObservable() {
    this.twitterStream = new TwitterStreamFactory().getInstance();
    this.twitterStream.addListener(new StatusListener() {
      @Override
      public void onStatus(Status status) {
        subscribers.forEach(s -> s.onNext(status));
      }

      @Override
      public void onException(Exception ex) {
```

```
          subscribers.forEach(s -> s.onError(ex));
        }

        //其他回调
      });
    }

    private final Observable<Status> observable = Observable.create(
        subscriber -> {
          register(subscriber);
          subscriber.add(Subscriptions.create(() ->
              this.deregister(subscriber)));
          });

    Observable<Status> observe() {
      return observable;
    }

    private synchronized void register(Subscriber<? super Status> subscriber) {
      if (subscribers.isEmpty()) {
        subscribers.add(subscriber);
        twitterStream.sample();
      } else {
        subscribers.add(subscriber);
      }
    }

    private synchronized void deregister(Subscriber<? super Status> subscriber) {
      subscribers.remove(subscriber);
      if (subscribers.isEmpty()) {
        twitterStream.shutdown();
      }
    }

  }
```

subscribers 的集线程安全地存储当前已订阅的 Observer 集合。每次新的 Subscriber 出现，将其添加到一个集中，并连接到底层的事件源上。相反，最后的 Subscriber 消失时，就关闭上游的源。这里的关键在于始终只有一个到上游系统的连接，而不是为每个订阅者都建立连接。这个实现能够正常运行而且比较健壮，但看起来过于低层级并且易于出错。对 subscribers 的访问必须使用 synchronized 同步，而且集合本身必须支持安全地迭代。对 register() 的调用必须发生在通过 deregister() 注销回调之前，否则，后者可能会在注册之前调用。将一个上游源多路复用到多个 Observer 的通用场景，肯定有更好的方式来实现。幸而，至少有两种这样的机制。RxJava 致力于减少这样危险的样板代码并抽象出并发性。

2.6 rx.subjects.Subject

Subject 类非常有意思，它扩展了 Observable，同时还实现了 Observer。这意味着既可以在客户端将其视为 Observable（订阅上游的事件），也可以在提供者端将其视为 Observer

（通过调用 onNext() 按需往下游推送事件）。通常情况下，我们的做法是在内部持有一个对 Subject 的引用，这样就可以从任何源推送事件，而对外则将该 Subject 暴露为 Observable。让我们使用 Subject 重新实现 Status 流的更新。为了进一步简化实现，立即连接到外部系统并且不再跟踪订阅者。除了能够简化样例之外，另一个优点就是能够减少第一个 Subscriber 出现时的延迟。如下所示，事件已经开始流动了，所以不需要等待重新连接到第三方应用程序上。

```
class TwitterSubject {

    private final PublishSubject<Status> subject = PublishSubject.create();

    public TwitterSubject() {
        TwitterStream twitterStream = new TwitterStreamFactory().getInstance();
        twitterStream.addListener(new StatusListener() {
            @Override
            public void onStatus(Status status) {
                subject.onNext(status);
            }

            @Override
            public void onException(Exception ex) {
                subject.onError(ex);
            }

            //其他回调
        });
        twitterStream.sample();
    }

    public Observable<Status> observe() {
        return subject;
    }

}
```

PublishSubject 是 Subject 的子类之一，会立即从上游系统接收事件，并简单地将它们推送（通过调用 subject.onNext(...)）至所有的 Subscriber。Subject 内部会跟踪这些事件，所以我们就没有必要这样做了。请注意，我们在 observe() 中返回了 subject，假装其为简单的 Observable。如果有人订阅，并且后台调用 onNext() 之后，Subscriber 会立即接收到后续的所有事件——至少在它取消订阅之前均是如此。Subject 在内部管理 Subscriber 的生命周期，所以调用 onNext() 即可，无须关心有多少订阅者在监听。

Subject 中的错误传播

Subject 非常有用，而且有很多细节需要理解。例如，在调用 subject.onError() 之后，Subject 会悄无声息地丢弃后续的 onError 通知，实际上就是将其吞噬掉了。

Observable.create(...) 看上去过于复杂、难以管理的时候，Subject 就是一个创建 Observable 实例的有用工具。其他类型的 Subject 如下所示。

- ❑ AsyncSubject

  记住发布的最后一个值，并且 onComplete() 被调用时，将其推送给所有的订阅者。如果 AsyncSubject 没有完成，除了最后一个事件之外，所有的事件都会被丢弃。

- ❑ BehaviorSubject

  在订阅之后，将所有发布的事件推送给订阅者，与 PublishSubject 类似。但是，它首先会发布在订阅之前发生的最新事件，从而让 Subscriber 立即得到流的状态。比如，Subject 可能会代表当前的温度，每分钟广播一次。从客户端订阅的时候，他将立即接收到最新的可见温度，而不用等待几秒再获得下一个事件。但是，同一个 Subscriber 只对最新的温度感兴趣，而不会对历史温度感兴趣。如果此时还没有发布过事件，并且提供了默认事件，那么将会首先推送一个特殊的默认事件。

- ❑ ReplaySubject

  最有意思的 Subject 类型会缓存贯穿整个历史的推送事件。如果有人订阅，他首先会接收到一批被错过（缓存）的事件，然后才会实时接收后续的事件。默认情况下，Subject 创建之后的所有事件都会被缓存。如果流是无穷的或者非常长，这可能会非常危险（参见 8.6 节）。在这种情况下，可以使用重载版本的 ReplaySubject，它只会保留如下事件。

  - 在内存中保留可配置数量的事件（createWithSize()）。
  - 配置最近事件的时间窗口（createWithTime()）。
  - 通过 createWithTimeAndSize() 同时限制大小和时间（限制较先达到的条件）。

Subject 要非常小心地使用：通常会有更惯用的方式来共享订阅和缓存事件，参见 2.7 节。目前，应该优先使用更低级的 Observable.create()，或者考虑标准的工厂方法，如 from() 和 just()。

另外一件需要记住的事情是并发。默认情况下，在 Subject 上调用 onNext() 会被直接传播至所有 Observer 的 onNext() 回调方法。这些方法使用相同的名字并不奇怪，从某种意义上说，在 Subject 上调用 onNext() 会间接调用所有 Subscriber 的 onNext()。但是，你需要记住，按照 Rx 设计指南，在 Observer 上对 onNext() 的所有调用必须都是序列化的（也就是串行），所以两个线程不能同时调用 onNext()。但是，你使用 Subject 的方式，可能很容易就会破坏这个规则，比如使用线程池中的多个线程来调用 Subject.onNext()。幸好，如果你担心这个问题，可以在 Subject 上只调用 .toSerialized()，它类似于调用 Observable.serialize()。这个操作符能够确保下游的事件都能按照正确的顺序出现。

2.7 ConnectableObservable

ConnectableObservable 以一种有意思的方式来协调多个 Subscriber，并共享一个底层的订阅。还记得最初借助 LazyTwitterObservable 创建单个延迟执行的对底层资源的连接吗？必须要手动跟踪所有的 Subscriber，如果第一个订阅者出现或最后一个订阅者离开，就建立连接或断开连接。ConnectableObservable 是 Observable 的一个特殊类型，能够确保底层始终最多只有一个 Subscriber，但实际上它又允许多个 Subscriber 共享相同的底层资源。

ConnectableObservable 有很多应用场景，例如：不管 Subscriber 何时订阅，都要确保它们接收到相同的事件序列。如果订阅时会有重要的副作用，即便还没有“真正”的 Subscriber，ConnectableObservable 也能强迫订阅。后续很快就将讨论上述的所有场景。Subject 是创建 Observable 的必要方式，而 ConnectableObservable 会保护原始的上游 Observable，并确保最多只能有一个 Subscriber 可以接触到它。不管有多少 Subscriber 连接到 ConnectableObservable，系统只会打开一个 Observable 的订阅，这个订阅是基于该 Observable 创建的。

2.7.1 使用publish().refCount()实现单次订阅

回顾一下：只有一个到底层资源的句柄，如到 Twitter 状态更新流的 HTTP 连接。但是，推送事件的这个 Observable 将会被多个 Subscriber 共享。前面创建的 Observable 的原生实现并没有对此进行控制，因此每个 Subscriber 会启动自己的连接。如下所示的程序代码是非常浪费的。

```
Observable<Status> observable = Observable.create(subscriber -> {
    System.out.println("Establishing connection");
    TwitterStream twitterStream = new TwitterStreamFactory().getInstance();
    //...
    subscriber.add(Subscriptions.create(() -> {
        System.out.println("Disconnecting");
        twitterStream.shutdown();
    }));
    twitterStream.sample();
});
```

尝试使用这个 Observable 的时候，每个 Subscriber 都会建立一个新的连接，如下所示。

```
Subscription sub1 = observable.subscribe();
System.out.println("Subscribed 1");
Subscription sub2 = observable.subscribe();
System.out.println("Subscribed 2");
sub1.unsubscribe();
System.out.println("Unsubscribed 1");
sub2.unsubscribe();
System.out.println("Unsubscribed 2");
```

以下是输出。

```
Establishing connection
Subscribed 1
Establishing connection
Subscribed 2
Disconnecting
Unsubscribed 1
Disconnecting
Unsubscribed 2
```

简单起见，这次使用一个无参数的 subscribe()，它会触发订阅，但是会将所有的事件和通知丢弃。本章几乎花费了一半的篇幅来解决这个问题，在这个过程中介绍了大量的 RxJava 特性。最后我们介绍一个最具有扩展性和最简单的解决方案：publish().refCount() 组合。

```
lazy = observable.publish().refCount();
//...
System.out.println("Before subscribers");
Subscription sub1 = lazy.subscribe();
System.out.println("Subscribed 1");
Subscription sub2 = lazy.subscribe();
System.out.println("Subscribed 2");
sub1.unsubscribe();
System.out.println("Unsubscribed 1");
sub2.unsubscribe();
System.out.println("Unsubscribed 2");
```

以下输出与预期非常相似。

```
Before subscribers
Establishing connection
Subscribed 1
Subscribed 2
Unsubscribed 1
Disconnecting
Unsubscribed 2
```

直到真正有第一个 Subscriber 的时候，连接才会建立。但是，更重要的在于，第二个 Subscriber 不会初始化新的连接，它甚至不会接触到原始的 Observable。publish().refCount() 会将底层的 Observable 串联包装起来，并拦截所有的订阅。稍后将会介绍为何需要两个方法，以及单独使用 publish() 意味着什么。暂时先关注 refCount()。这个操作符基本就是记录此刻有多少活跃的 Subscriber，非常类似于历史上的垃圾 – 收集算法。这个数字从零变成一的时候，它会订阅上游的 Observable。所有超过一的数字都会被忽略，同一个上游 Subscriber 就被所有的下游 Subscriber 共享。但是，如果最后一个下游 Subscriber 取消订阅，计数器将会从一减少到零，refCount() 就知道必须马上取消订阅了。refCount() 完全做到了通过 LazyTwitterObservable 手动实现的功能。使用 publish().refCount() 组合以共享同一个 Subscriber，同时依然保持延迟执行的特征。这对操作符经常同时使用，因此有了一个别名 share()。需要记住的是，如果订阅之后很快就取消订阅，share() 依然会执行重新连接，就像没有缓存一样。

2.7.2 ConnectableObservable的生命周期

publish() 操作符另外一个有效的用例就是在没有任何 Subscriber 的情况下，强制进行订阅。假设有一个自己的 Observable<Status>，在将它暴露给客户端之前，不管有没有人订阅，我们都想要把每个事件存储在数据库中。如下所示，最原始的方案是不够的。

```
Observable<Status> tweets = //...
return tweets
    .doOnNext(this::saveStatus);
```

这里使用 doOnNext() 操作符，它会探查经过流的每一个条目并执行一些操作，比如 saveStatus()。但是，不要忘记，根据设计 Observable 是延迟执行的。因此，如果没有人订阅，doOnNext() 不会触发。这里想要的是一个假的 Observer，它不会真正地监听事件，但是能够强迫上游的 Observable 生成事件。一个重载版本的 subscribe() 就可以完成这项

任务，如下所示。

```
Observable<Status> tweets = //...
tweets
    .doOnNext(this::saveStatus)
    .subscribe();
```

这个空的 Subscriber 最终会触发 Observable.create()，并建立到上游事件源的连接。看上去，这样已经解决了这个问题，但是依然无法解决多个订阅者带来的问题。如果将 tweets 暴露出去，第二个订阅者将会尝试发起对外部资源的第二次连接，比如打开第二个 HTTP 连接。惯用的解决方案是组合使用 publish().connect()，它们会立即创建一个人工的 Subscriber，同时还能保持只有一个上游的 Subscriber。最好还是使用一个例子来进行阐述。最后，我们还将介绍单独的 publish() 是如何运行的，如下所示。

```
ConnectableObservable<Status> published = tweets.publish();
published.connect();
```

最终，我们看到了 ConnectableObservable 的所有功能。可以在任意 Observable 上调用 Observable.publish()，并且返回一个 ConnectableObservable。在这里，可以继续使用原始的上游 Observable（前面样例中的 tweets），publish() 并不会影响到它。但是，现在我们更关注返回的 ConnectableObservable。订阅 ConnectableObservable 的人会被放到一个 Subscriber 集合。如果没有调用 connect()，这些 Subscriber 只是被搁置在那里，它们不会直接订阅上游的 Observable。但是，如果调用了 connect()，一个专用的中间 Subscriber 将会订阅上游的 Observable（即 tweets），不管之前出现过多少下游的订阅者，即便没有任何的下游订阅者。但是如果 ConnectableObservable 的一些 Subscribe 搁置了，那么它们都会接收到相同序列的通知。

这种机制有很多优势。假设在应用中有一个 Observable，多个 Subscriber 都对其感兴趣。在启动的时候，会有多个组件（如 Spring bean 或 EJB）订阅该 Observable 并开始监听。如果没有 ConnectableObservable，hot 类型的 Observable 开始发布的事件很可能只能由第一个 Subscriber 消费，随后启动的 Subscriber 会丢失早期的这些事件。如果你希望所有的 Subscriber 收到系统的一致视图，那可能很困难。所有的 Subscriber 都会以相同的顺序接收到事件，但是晚出现的 Subscriber 将会错失早期的通知。

这个问题的解决方案就是先将这个 Observable 通过 publish() 发布出去，这样系统中的所有组件就都能借助 subscribe() 进行订阅，比如在系统启动的过程中完成订阅。如果 100% 确定需要接收相同事件序列（包括最初的事件）的所有 Subscriber 都能够进行 subscribe() 操作了，那就使用 connect() 连接 ConnectableObservable。这将会创建一个上游 Observable 的 Subscriber，并将事件推送给所有下游的 Subscriber。如下的样例使用了 Spring 框架，但代码本身其实是与框架无关的。

```
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.event.ContextRefreshedEvent;
import rx.Observable;
import rx.observables.ConnectableObservable;
```

```java
@Configuration
class Config implements ApplicationListener<ContextRefreshedEvent> {

    private final ConnectableObservable<Status> observable =
        Observable.<Status>create(subscriber -> {
            log.info("Starting");
            //...
        }).publish();

    @Bean
    public Observable<Status> observable() {
        return observable;
    }

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        log.info("Connecting");
        observable.connect();
    }
}

@Component
class Foo {

    @Autowired
    public Foo(Observable<Status> tweets) {
        tweets.subscribe(status -> {
            log.info(status.getText());
        });
        log.info("Subscribed");
    }
}

@Component
class Bar {

    @Autowired
    public Bar(Observable<Status> tweets) {
        tweets.subscribe(status -> {
            log.info(status.getText());
        });
        log.info("Subscribed");
    }
}
```

这个简单的程序首先立即创建了一个 Observable（底层是 ConnectableObservable 子类）。按照设计，Observable 是延迟执行的，所以甚至可以静态地创建它们。这个 Observable 通过 publish() 进行了发布，所以，在进行 connect() 之前，所有后续的 Subscriber 都会被搁置起来，不会收到任何的通知。随后，框架会发现两个带有 @Component 注解的组件，它们需要这个 Observable。依赖注入框架会提供 ConnectableObservable 并允许任何人进行订阅。但是，在程序完整启动之前，事件也不会出现，即便是 hot 类型的 Observable。所有的组件都实例化并装配完成之后，可以消费由框架发送的 ContextRefreshedEvent 事件。

此时，可以确保所有的组件都能请求指定的 Observable，并且通过 subscribe() 订阅。程序马上启动的时候，调用 connect()。这样只对底层 Observable 进行一次订阅，完全相同的事件序列会转发给每个组件。裁剪之后的日志输出如下所示（方括号中对应的是组件的名称）。

```
[Foo    ] Subscribed
[Bar    ] Subscribed
[Config] Connecting
[Config] Starting
[Foo    ] Msg 1
[Bar    ] Msg 1
[Foo    ] Msg 2
[Bar    ] Msg 2
```

注意，Foo 和 Bar 组件报告了它们订阅成功的信息，即使还没有收到任何的事件。只有在应用程序完全启动之后，connect() 才会订阅底层的 Observable 并将 Msg 1 和 Msg 2 转发给所有的组件。相同场景下与简单 Observable 进行对比，如果不使用 ConnectableObservable，并允许每个组件立即订阅，那么输出可能会如下所示。

```
[Config] Starting
[Foo    ] Subscribed
[Foo    ] Msg 1
[Config] Starting
[Bar    ] Subscribed
[Foo    ] Msg 2
[Bar    ] Msg 2
```

这里有两个差异需要注意。首先，Foo 组件订阅的时候，它会立即开启一个到底层资源的连接，它并不会等待应用启动完成。更糟糕的是，Bar 组件会初始化另外一个连接（注意 Starting 出现了两次）。其次，Bar 组件是从 Msg 2 开始的，并没有收到 Msg 1，这条信息只被 Foo 组件接收到了。消费 hot 类型的 Observable 时的不一致性，在有些场景下可能是问题，也可能不是什么问题，但是你必须要注意。

2.8 小结

创建和订阅 Observable 是 RxJava 的重要特性。尤其是很多初学者会忘记订阅，于是对没有事件发布出来感到意外。很多开发人员关注这个库提供的酷炫的操作符（参见第 3 章），但是如果无法理解这些操作符在底层如何执行订阅，将会导致一些微妙的缺陷。

另外，RxJava 的异步特性通常被认为是理所当然的，但事实并非如此。实际上，RxJava 中的大多数操作符并没有使用任何特殊的线程池。更确切地说，这意味着默认情况下根本就不会涉及并发，所有的操作都会在客户端线程中执行。这是本章另外一个核心要点。现在，你已经理解了订阅和并发原则，能够更加轻松和高效地使用 RxJava 了。

第 3 章将介绍如何使用这个库提供的内置操作符，以及如何将它们组合起来。声明式转换和流组合的特性让 RxJava 获得了空前的关注。

第 3 章

操作符与转换

托马什·努尔凯维茨（Tomasz Nurkiewicz）

本章的目标在于阐述 RxJava 操作符（operator）的基础知识，以及如何组合它们构建高层级、易于理解的数据管道。RxJava 强大的一个原因是它提供了丰富的内置操作符，并且还支持自定义操作符。操作符是一个函数，它接受上游的 Observable<T> 并为下游返回 Observable<R>，这里的类型 T 和 R 可能相同也可能不同。操作符可以将简单的转换组合为复杂的处理图。

例如，Observable.filter() 操作符从上游的 Observabl 接受条目，但是只会转发匹配指定断言的条目。与之不同的是，Observable.map() 会转换经过它的条目。这样就可以提取、填充（enrich）或包装原有的事件。有些操作符要更为复杂。比如，Observable.delay() 会原样将事件传递出来，但是在一个固定延迟之后所有事件才会出现。最后，还有一些操作符（如 Observable.buffer()）可能以成批处理的方式，在消费一些输入之后才会将其发布出来。

你也许已经意识到单个 Rx 操作符的优势，但是它真正的强大之处在于组合。链接多个操作符，将流分成多个子流，然后再将它们联合起来，这是习惯性的做法。经过这一章的学习，你应该能够对此驾轻就熟。

3.1　核心的操作符：映射和过滤

操作符是 Observable 典型的实例方法，它会按照某种形式变更上游 Observable 的行为，下游的 Observable 或 Subscriber 看到的是变化后的结果。这听上去可能有些复杂，但是它实际上非常灵活，并且理解起来没有那么困难。操作符的最简单样例就是 filter()，它会接受一个断言，事件要么能够通过，要么会被丢弃。

```
Observable<String> strings = //...
Observable<String> filtered = strings.filter(s -> s.startsWith("#"));
```

现在开始介绍所谓的**弹珠图**（marble diagram），它是在 RxJava 中普遍存在的一种可视化形式。弹珠图阐述了各种操作符是如何运行的。大多数情况下，你会看到两个水平轴，代表了时间从左向右推移。图中的形状（前文提到的弹珠）对事件实现了可视化。上面的轴线和下面的轴线之间就是需要讲解的操作符，该操作符会以某种形式改变来自源 `Observable`（上游）的事件序列，并形成最后的 `Observable`（下游），如图 3-1 所示。

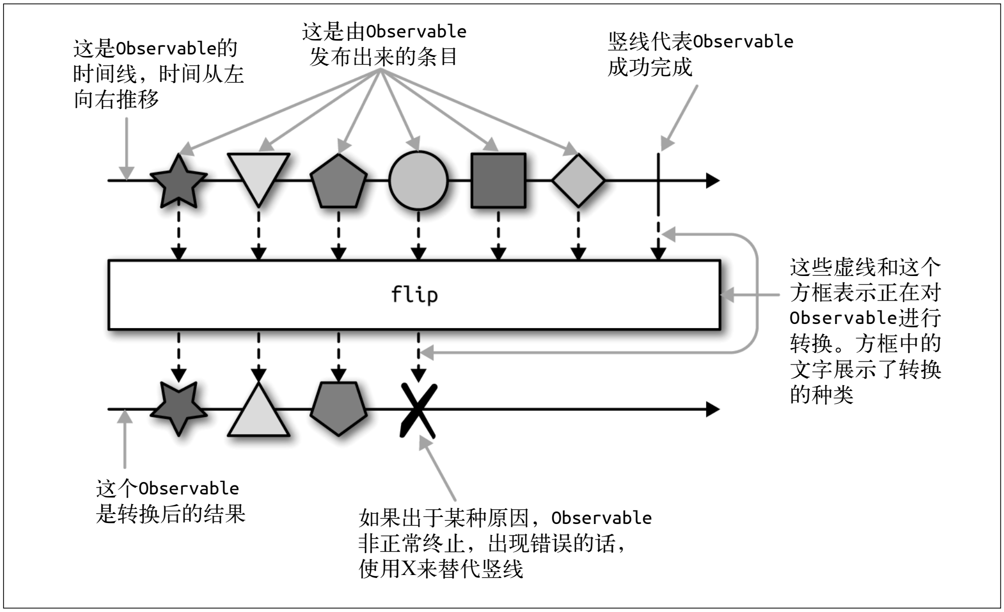


图 3-1

图 3-2 展现了弹珠图的一个具体样例，它阐释了 `filter()` 操作符。`Observable.filter()` 返回完全相同的事件（所以上面和下面的弹珠形状完全相同），但是有些事件因为不满足断言被过滤掉了。

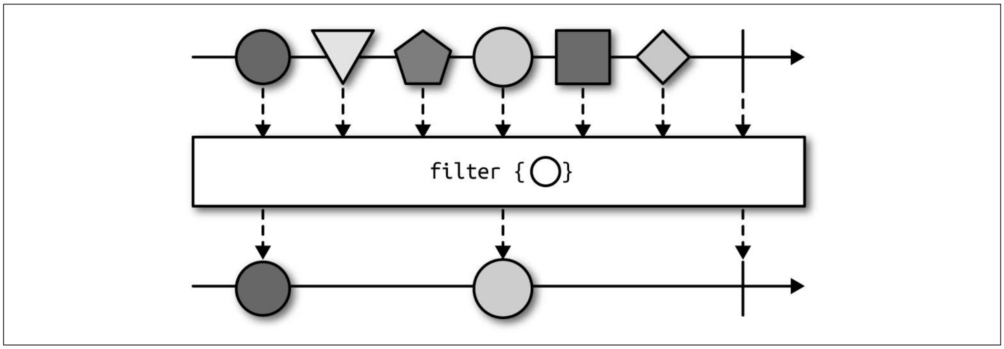


图 3-2

在处理特定类型的 Observable 时，你可能对某些事件并不感兴趣，比如处理大量数据。对同一个 Observable 进行多次 filter() 操作也是常见的做法，每次过滤会使用不同的断言。我们可以对原始的 Observable 使用多个过滤器，甚至还可以将它们链接起来（filter(p1).filter(p2).filter(p3)），从而实现逻辑联结（filter(p1 && p2 && p3)）。将多个操作符（不是仅适用于 filter()）合并为一个既有优势也有不足。如果你能够以不同的方式重用更小的转换或组合它们的话，那么推荐使用更小的转换（如多次过滤）。而另一方面，更多的操作符会增加开销[1]和栈的深度。选择哪种形式取决于你的需求和代码风格。

```
Observable<String> strings = someFileSource.lines();
Observable<String> comments = strings.filter(s -> s.startsWith("#"));
Observable<String> instructions = strings.filter(s -> s.startsWith(">"));
Observable<String> empty = strings.filter(String::isBlank);
```

你可能会问自己这样一个问题：上游的原始 strings 源发生了什么呢？基于面向对象的背景，你可能会记得像 java.util.List.sort() 这样的方法，它会在 List 内部对条目进行重新排序，并且什么内容都不返回。Java 的 List<T> 是可变的（有利也有弊），所以对它的内容进行重新排序是可以的。类似地，我们可以假想一个 void List.filter() 方法，它接受一个断言，在内部移除不匹配的元素。在 RxJava 中，你必须要忘掉内部可变的数据结构：在流之外修改变量通常被视为不符合常规且危险的。每个操作符都要返回一个**全新**的 Observable，而原有的 Observable 要保持不变。

这会让事件流的原理更加简单。你可以将一个流分成多个独立的源，每个源都具有不同的特征。RxJava 的一个强大之处在于，你可以在多个地方重用同一个 Observable，而不会影响到其他的消费者。如果你将 Observable 传递给一个未知的函数，你可以确保这个 Observable 不会被该函数以任何方式破坏掉。对于可变的 java.util.Date，你无法做出这样的保证，因为任何引用它的人都可以对其进行修改。这也是新的 java.time API 完全不可变的原因。

3.1.1 使用map()进行一对一转换

假设有一个用事件组成的流，你必须要对每个事件进行特定的转换。这可能是从 JSON 解码为 Java 对象（反之亦然），填充，包装，从事件中进行抽取，等等。这就是重要的 map() 操作符能够发挥作用的地方。它会对来自上游的每个值进行转换，如下所示。

```
import rx.functions.Func1;


Observable<Status> tweets = //...
Observable<Date> dates = tweets.map(new Func1<Status, Date>() {
    @Override
    public Date call(Status status) {
        return status.getCreatedAt();
    }
});
```

注 1：正在开展操作符融合的相关研究，期望将多个操作符无缝合并为一个。

```
Observable<Date> dates =
        tweets.map((Status status) -> status.getCreatedAt());

Observable<Date> dates =
        tweets.map((status) -> status.getCreatedAt());

Observable<Date> dates =
        tweets.map(Status::getCreatedAt);
```

定义 `dates Observable` 的方式其实都是等价的，从最烦琐的 `Func1<T, R>` 到 Java 8 语法中最紧凑的方法引用和类型推断。但是，请看仔细！名为 `tweets` 的原始 `Observable` 生成的是 `Status` 类型的事件。随后，调用 `map()`，通过一个函数将单个事件（`Status s`）转换为 `Date` 类型的返回值。顺便提一下，可变事件（如 `java.util.Date`）是有问题的，因为可变事件被其他 `Subscribers` 消费时，任意的操作符或 `Subscriber` 都可能无意间改变这些事件。可以使用后续的 `map()` 快速修正该问题，如下所示。

```
Observable<Instant> instants = tweets
        .map(Status::getCreatedAt)
        .map((Date d) -> d.toInstant());
```

`map()` 的弹珠图，如图 3-3 所示。

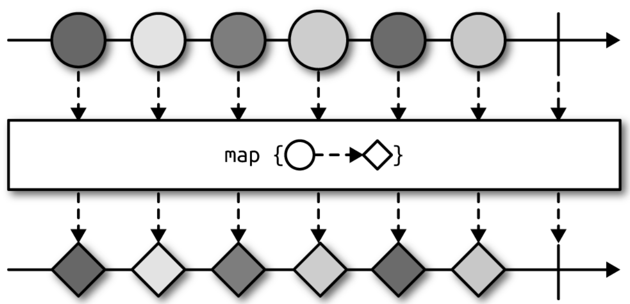


图 3-3

`map()` 操作符接收一个函数，该函数能够将输入事件的形状从圆形改为方形。这种转换会应用到流经的每个条目上。

接下来是一项突击测验，确保你真正理解了 `Observable` 是如何运行的。观察如下代码，然后预测一下 `Observable` 被订阅时，将会发布出什么样的值。

```
Observable
        .just(8, 9, 10)
        .filter(i -> i % 3 > 0)
        .map(i -> "#" + i * 10)
        .filter(s -> s.length() < 4);
```

`Observable` 是**延迟**执行的，这意味着只在有人订阅的情况下，它们才会产生事件。你可以创建一个无穷流，耗费几个小时来生成第一个值。但是，在你实际表明想要订阅这些事件的通知之前，`Observable` 只是一个被动和空闲的代表 `T` 类型的数据结构。这种情况甚至也

适用 hot 类型的 Observable。即便事件源在不断地生成事件，但是在没有人真正表示感兴趣之前，像 map() 或 filter() 这样的操作符都不会被执行。否则，哪怕运行所有的计算步骤并将结果抛出来，也没有任何的意义。每次使用操作符的时候，包括本书还未介绍的那些，实际上就是围绕着原始的 Observable 创建了一个包装器。这个包装器能够拦截所有经过的事件，但是它本身并不会进行订阅。

```
Observable
        .just(8, 9, 10)
        .doOnNext(i -> System.out.println("A: " + i))
        .filter(i -> i % 3 > 0)
        .doOnNext(i -> System.out.println("B: " + i))
        .map(i -> "#" + i * 10)
        .doOnNext(s -> System.out.println("C: " + s))
        .filter(s -> s.length() < 4)
        .subscribe(s -> System.out.println("D: " + s));
```

消息经过流时，对它们进行日志记录和窥探是非常有用的。有一个特殊的**非纯粹**（impure）的操作符，名为 doOnNext()，可以在不接触这些条目的情况下对其进行查看。说它不纯粹是因为必须要依赖于副作用，比如记录日志或访问全局状态。doOnNext() 只是简单地接收来自上游 Observable 的每个事件并将其传递给下游，它不能以任何方式对事件进行修改。doOnNext() 类似一个探针，可以注入 Observable 管道的任何地方，从而观察流经管道的内容。这是侦听（wiretap）模式的简单实现，该模式可以在 Hohpe 和 Woolf 编著的《企业集成模式：设计、构建及部署消息传递解决方案》中找到。从技术上讲，doOnNext() 可以对事件做出改变。但是，Observable 控制的可变事件将会导致灾难性的后果。你很快将会学习如何并发处理事件，以及 fork 执行，等等。保证每个事件的线程安全是关键。根据经验，对于所有实际的应用程序，Observable 包装的所有类型都应该是不可变的。

首先，了解一下 RxJava 的执行路径。上述代码样例中的每一行都会通过包装原有的 Observable 创建一个新的 Observable。例如，第一个 filter() 并不会从 Observable.just(8, 9, 10) 中将 9 移除，相反，它会创建一个新的 Observable，如果订阅这个新的 Observable，最终发布的值是 8 和 10。相同的原则适用于大多数操作符：它们不会修改已有 Observable 的内容和行为，而会创建新的 Observable。但是，说 filter() 或 map() 只是**创建**新的 Observable 有点太简略了。大多数的操作符都是延迟执行的，直到有人订阅它们才会执行。那么，Rx 在操作链的末尾看到 subscribe() 会发生些什么呢？理解内部能够帮助你掌握流在底层是如何进行处理的。让我们从下往上看一下这些代码。

- 首先，subscribe() 通知上游 Observable 它想接收值。
- 上游的 Observable（filter(s -> s.length() < 4)）本身并没有任何的条目，它只是围绕另一个 Obervable 的装饰器。所以，它也会订阅上游的流。
- 同 filter() 类似，map(i -> "#" + i * 10) 本身并不会投递任何的条目，它只对收到的任意内容进行转换。因此，它必须像其他操作符那样订阅上游的流。
- 这会一直持续到抵达 just(8, 9, 10)。这个 Observable 是真正的事件源。filter(i -> i % 3 > 0) 订阅它的时候（这是后面显式 subscribe() 的一个结果），它就会开始为下游推送事件。

- 现在，可以观察一下事件是如何流经管道中的所有阶段的。filter() 内部接收到 8 并将其传递到下游（i % 3 > 0 断言为真）。随后，map() 将 8 转换为字符串 "#80"，然后激活它下面的 filter() 操作符。
- 断言 s.length() < 4 同样可行，然后可以将转换后的值传递给 System.out。

doOnNex() 版本生成的输出如下，你可以花些时间研究一下 9 和 10 是如何从结果中被剔除的。

```
A: 8
B: 8
C: #80
D: #80
A: 9
A: 10
B: 10
C: #100
```

3.1.2 使用flatMap()进行包装

flatMap() 是 RxJava 中最重要的操作符之一。乍看上去，它类似于 map()，但是它对每个元素的转换都会返回另外一个（内嵌的）Observable。鉴于 Observable 可以代表另外一个异步操作，我们很快就意识到 flatMap() 可以为上游的每个事件执行异步计算（fork 执行）并将结果加入进来。从概念上讲，flatMap() 接收一个 Observable<T> 以及一个从 T 到 Observable<R> 类型的函数。flatMap() 首先会构造一个 Observable<Observable<R>>，将上游 T 类型的值替换为 Observable<R>（与 map() 类似）。但是，并未结束，它会自动订阅这些内部的 Observable<R> 流，并生成一个 R 类型的流，这个流包含了所有内部流的值。图 3-4 的弹珠图展现了它是如何运行的。

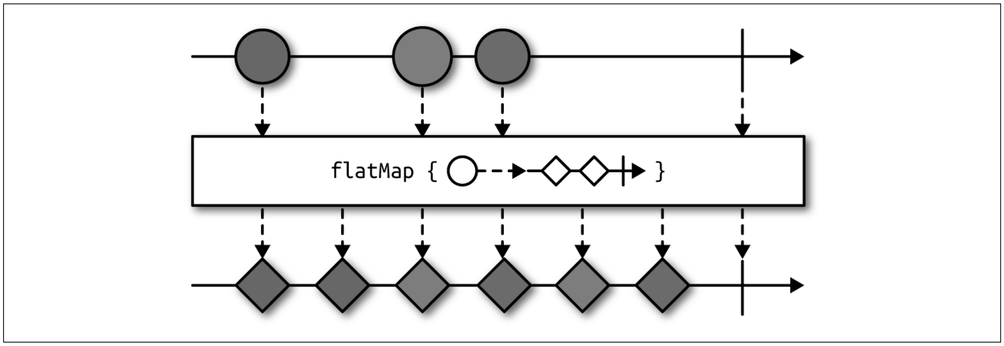


图 3-4

上面的弹珠图涉及了 flatMap() 的一个重要方面。每个上游的事件（圆形）会转换成两个菱形组成的 Observable，这两个菱形之间有一定的时间间隔。如果两个上游事件出现的时间非常接近，flatMap() 会自动进行转换并将它们变成两个菱形的流。但是，因为 RxJava 会并发地对它们进行订阅并将结果合并在一起，所以某个内部 Observable 生成的事件可能会与其他 Observable 生成的事件交叉在一起。稍后会探讨这种行为。

flatMap() 是 RxJava 最基础的操作符之一，能够很容易地[2] 实现 map() 或 filter()。

```
import static rx.Observable.empty;
import static rx.Observable.just;

numbers.map(x -> x * 2);
numbers.filter(x -> x != 10);

//两者是等价的
numbers.flatMap(x -> just(x * 2));
numbers.flatMap(x -> (x != 10) ? just(x) : empty());
```

先来看 flatMap() 的一个更实际的样例。假设你会接收到汽车进入高速公路的一系列照片。对于每一辆汽车，会运行一个代价比较高的光学字符识别算法，它会根据汽车的牌照返回相应的注册号码。显然，识别过程可能会失败，在这种情况下，该算法将不会返回任何内容。它还可能会因为异常而失败，或者某些诡异的问题造成一辆车返回两个牌照。通过 Observable 可以很容易地为其建模。

```
Observable<CarPhoto> cars() {
    //...
}

Observable<LicensePlate> recognize(CarPhoto photo) {
    //...
}
```

将 Observable<LicensePlate> 作为基础的数据流，你可以对其建模以适应如下场景。

- 在照片中获取不到牌照（空流）。
- 发生致命的内部故障（onError() 回调）。例如，识别模块完全地、永久地失败，而且没有恢复方案。
- 识别出一个或多个牌照，随后是 onComplete()。

更棒的是，随着时间的推移，recognize() 还可以稳定地产生更好的效果，比如从粗略估计或并发运行两个算法开始。如下是使用了上述方法的示例代码。

```
Observable<CarPhoto> cars = cars();

Observable<Observable<LicensePlate>> plates =
        cars.map(this::recognize);

Observable<LicensePlate> plates2 =
        cars.flatMap(this::recognize);
```

从 map() 函数返回的内容会再被包装在一个 Observable 中。这意味着，如果你返回 Observable<LicensePlate> 的话，实际得到的将会是 Observable<Observable<LicensePlate>>。一个 Observable 嵌入到另一个 Observable，不仅使用起来非常麻烦，而且为了获取结果，还必须首先订阅每个内部的 Observable。除此之外，你还需要以某种方式将内部结果同步到一个流中，这是非常困难的。

注 2：然而考虑到性能，RxJava 有专用的 map() 和 filter() 实现。

flatMap() 通过将结果**扁平化**（flattening）解决了这个问题，这样得到的就是一个简单的 LicensePlate 流。除此之外，4.9 节将会介绍如何使用 flatMap() 运行并行任务。一般可以将 flatMap() 用于如下场景。

- map() 转换的结果必须是 Observable。比如，对流中的每个元素执行长时间运行的异步操作，这些操作是非阻塞的。
- 需要进行一对多的转换，一个事件要被扩展为多个子事件。例如，客户组成的流要被转换为他们的订单组成的流，而每个客户可能会有任意数量的订单。

现在，假设你想要使用一个返回 Iterable（如 List 或 Set）的方法。例如，Customer 有一个很简单的 List<Order> getOrders() 方法，要将其用到 Observable 管道中，必须通过一些操作符，如下所示。

```
Observable<Customer> customers = //...
Observable<Order> orders = customers
        .flatMap(customer ->
            Observable.from(customer.getOrders()));
```

或者使用如下的等价形式，不过依然很烦琐。

```
Observable<Order> orders = customers
        .map(Customer::getOrders)
        .flatMap(Observable::from);
```

将一个条目映射到 Iterable 的需求非常普遍，因此 RxJava 创建了一个专门的操作符 flatMapIterable() 来执行这种转换，如下所示。


```
Observable<Order> orders = customers
        .flatMapIterable(Customer::getOrders);
```

简单地包装 Observable 中的方法时，你必须非常小心。如果 getOrders() 并不是一个简单的 getter 方法，反而在运行时间方面代价高昂，那么最好重新实现 getOrders()，从而显式返回 Observable<Order>。

flatMap() 的另外一个变种不仅能够对事件做出反应，还能对所有通知做出反应，也就是事件、错误通知和完成通知。随后，这个 flatMap() 简化签名进行重载。对于每个 Observable<T>，必须要提供如下内容。

- 映射单个 T → Observable<R> 的函数。
- 映射错误通知 → Observable<R> 的函数。
- 无参函数，响应上游的完成通知并能够返回 Observable<R>。

代码如下所示。

```
<R> Observable<R> flatMap(
        Func1<T, Observable<R>> onNext,
        Func1<Throwable, Observable<R>> onError,
        Func0<Observable<R>> onCompleted)
```

假设你正在创建一个上传视频的服务。该服务接收一个 UUID，并通过 Observable<Long> 返回上传进度，也就是传输了多少字节。可以按照某些方式使用这个进度信息，比如在用户

界面对其进行展现。但是，我们真正关心的是它的完成信息，也就是上传何时最终完成。只有在成功上传之后，才能对视频进行评级。原生实现可能只会订阅进度流，而忽略事件并且只对完成通知（最后一个回调）进行响应，如下所示。

```
void store(UUID id) {
    upload(id).subscribe(
            bytes -> {}, //忽略
            e -> log.error("Error", e),
            () -> rate(id)
    );
}

Observable<Long> upload(UUID id) {
    //...
}

Observable<Rating> rate(UUID id) {
    //...
}
```

但是，需要注意 `rate()` 方法实际返回的 `Observable<Rating>` 丢失了，而我们真正想要的是 `store()` 返回第二个 `Observable<Rating>`。不过，不能简单地并发调用 `upload()` 和 `rate()`。因为如果前者尚未完成，后者将会失败。解决问题的答案依然是 `flatMap()`，不过要使用最复杂的形式。

```
upload(id)
        .flatMap(
                bytes -> Observable.empty(),
                e -> Observable.error(e),
                () -> rate(id)
        );
```

花些时间来分析一下上面的代码。这里有一个 `upload()` 方法返回得到的 `Observable<Long>`。对于 `Long` 类型值的每个进度更新，都返回 `Observable.empty()`，实际上也就是丢弃了这些事件。我们并不关心进度指示器的值，同样，也不关心错误信息。不仅不记录错误信息，反而将其传递给订阅者。需要注意，原生方式只是简单地记录日志错误，实际上是隐藏了它们。一般而言，如果不知道如何处理异常，那么就让你的监管者（如调用方法、父任务或下游的 `Observable`）来做出决策。最后的 lambda 表达式（`() -> rate(id)`）会对流的完成通知做出反应。此时，将完成通知替换为另外一个 `Observable<Rating>`。所以，即便原始的 `Observable` 想要终止，也会忽略它并以某种方式生成一个不同的 `Observable`。需要注意，这三个回调必须返回具有相同类型 `R` 的 `Observable<R>`。

在实践中，基于代码清晰性和性能方面的考虑，并不会使用 `flatMap()` 替换 `map()` 和 `filter()`。为了确保你已经理解了 `flatMap()` 的语义，以下是另外一个抽象的例子，它会将一个字符序列转换为摩斯码（morse code）。

```
import static rx.Observable.empty;
import static rx.Observable.just;

Observable<Sound> toMorseCode(char ch) {
    switch(ch) {
```

```
            case 'a': return just(DI, DAH);
            case 'b': return just(DAH, DI, DI, DI);
            case 'c': return just(DAH, DI, DAH, DI);
            //...
            case 'p': return just(DI, DAH, DAH, DI);
            case 'r': return just(DI, DAH, DI);
            case 's': return just(DI, DI, DI);
            case 't': return just(DAH);
            //...
            default:
                return empty();
        }
    }

enum Sound { DI, DAH }

//...

just('S', 'p', 'a', 'r', 't', 'a')
    .map(Character::toLowerCase)
    .flatMap(this::toMorseCode)
```

你可以清晰地看到，每个字符都被替换成了一个 DI 和 DAH 声音（**点**和**划线**）的序列。如果字符无法识别，将会返回一个空的序列。flatMap() 确保能够得到一个稳定、扁平化的声音流。而如果使用简单的 map()，得到的则是 Observable<Observable<Sound>>。此时，涉及 flatMap() 的一个重要方面，那就是事件的顺序。这最好通过一个例子来进行阐述，如果使用 delay() 操作符，这个例子可能会更有意思。

3.1.3 使用delay()操作符延迟事件

delay() 操作符只会接收上游的 Observable，并在一定的时间后发布所有的事件。所以，它的构造很简单，如下所示。

```
import java.util.concurrent.TimeUnit;


just(x, y, z).delay(1, TimeUnit.SECONDS);
```

在订阅之时，它并不会立即发布 x、y 和 z，而是在给定延迟之后再发布。

第 2 章已经介绍了 timer() 操作符，它们非常相似。可以使用 timer() 和 flatMap() 来替代 delay()，如下所示。

```
Observable
    .timer(1, TimeUnit.SECONDS)
    .flatMap(i -> Observable.just(x, y, z))
```

以上代码通过 timer() 生成了一个人为事件，但是我们完全忽略掉了。然而，使用 flatMap() 将这个人为事件（也就是 i 值中保存的零）替换为三个立即发布的值：x、y 和 z。在这个特殊场景下，可以说这与 just(x, y, z).delay(1, SECONDS) 是等价的。但是，一般并非如此，delay() 要比 timer() 的功能更加强大，它会将每个事件都放到给定的时间之后再发布出来，而 timer() 只是进行“休眠”，在给定的时间之后发布一个特定的事件。为了

完整，再看一下 delay() 的一个重载变种形式，它会计算单个事件的延迟，而不是全局所有事件。如下代码片段会延迟每个 String 值的发布，其延迟时间由 String 的长度决定。

```
import static rx.Observable.timer;
import static java.util.concurrent.TimeUnit.SECONDS;

Observable
    .just("Lorem", "ipsum", "dolor", "sit", "amet",
          "consectetur", "adipiscing", "elit")
    .delay(word -> timer(word.length(), SECONDS))
    .subscribe(System.out::println);

TimeUnit.SECONDS.sleep(15);
```

在运行这个程序的时候，即便进行了订阅，应用程序也会立即终止而不展现任何的结果，这是因为事件的发布是在后台运行的。第 4 章将会介绍 BlockingObservable，它会让这样简单的测试更加容易。但是，目前只需要在最后添加任意一个 sleep()。你会发现，出现的第一个单词是 sit，1 秒后，出现的是 amet 和 elit。还记得 delay() 可以通过 timer() 和 flatMap() 进行重写吗？你可以自己尝试一下，解决方法如下所示。

```
Observable
    .just("Lorem", "ipsum", "dolor", "sit", "amet",
          "consectetur", "adipiscing", "elit")
    .flatMap(word ->
        timer(word.length(), SECONDS).map(x -> word))
```

上述样例暴露了 flatMap() 一个很有意思的特点：它并不能保证原始事件的顺序。了解 delay() 如何运行之后，终于可以解决这个问题了。

3.1.4 flatMap()之后的事件顺序

从本质上来讲，flatMap() 接收一个随时间（事件）出现的值的**主**（master）序列（Observable），然后将每个事件分别替换为独立的子序列。这些子序列彼此之间是不相关的，并且与生成它们的主序列中的事件也是不相关的。更确切地说，此时拥有的不再是单个主序列，而是一组 Observable，其中每个都是独立运行的，并且随着时间的推移出现和消失。因此，flatMap() 并不能对子事件抵达下游操作符 / 订阅者的顺序给出**任何**的保证。以下面的简单代码片段为例。

```
just(10L, 1L)
    .flatMap(x ->
        just(x).delay(x, TimeUnit.SECONDS))
    .subscribe(System.out::println);
```

这个样例将事件 10L 延迟了 10 秒，又将事件 1L（按时间顺序，在上游的流中它是稍后出现的事件）延迟了 1 秒。结果就是，会在 1 秒之后看到 1，并且再过 9 秒看到 10——上游和下游事件的顺序是不同的！更糟糕的是，假设一个 flatMap() 转换要在很长的时间范围内生成多个（甚至是无穷个）事件。

```
Observable
        .just(DayOfWeek.SUNDAY, DayOfWeek.MONDAY)
        .flatMap(this::loadRecordsFor);
```

`loadRecordsFor()` 方法根据星期几返回不同的流。

```
Observable<String> loadRecordsFor(DayOfWeek dow) {
    switch(dow) {
        case SUNDAY:
            return Observable
                .interval(90, MILLISECONDS)
                .take(5)
                .map(i -> "Sun-" + i);
        case MONDAY:
            return Observable
                .interval(65, MILLISECONDS)
                .take(5)
                .map(i -> "Mon-" + i);
        //...
    }
}
```

`loadRecordsFor()` 中的一些重复是有意为之：提升这个日益复杂的样例的可读性。尽管如此，我们依然会逐步学习这个 `flatMap()`。这是一个简单的 `Observable`，会发布星期几的信息：星期日（Sunday）之后马上就是星期一（Monday）。现在，使用 `interval()` 生成的子序列对这些值进行转换。快速提示一下，`interval()` 将会按照固定的延迟，从零开始生成不断递增的数字。样例中的延迟取决于星期几：星期日（Sunday）和星期一（Monday）分别为 90 毫秒和 65 毫秒。每个序列均只取前 5 个条目（`take(5)`，参见 3.2 节），最终同时得到了两个 `Observable`，它们具有不同的频率。你预期得到的输出是什么呢？最简单直接的答案可能如下所示。

```
Sun-0, Sun-1, Sun-2, Sun-3, Sun-4, Mon-0, Mon-1, Mon-2, Mon-3, Mon-4
```

实际上，你有两个独立运行的流，但是它们的结果必须以某种形式**合并**（merge）到一个 `Observable`。`flatMap()` 遇到上游中星期日（Sunday）时，它会立即调用 `loadRecordsFor(Sunday)`，并将该函数结果（`Observable<String>`）的所有事件转发到下游。但是，几乎与此同时，星期一（Monday）会出现，`flatMap()` 将会调用 `loadRecordsFor(Monday)`。后面的子流产生的事件也会被传递到下游，与第一个子流产生的事件会交叉在一起。如果 `flatMap()` 想要避免重叠，它要么缓冲所有后续的子 `Observable`，直到第一个子 `Observable` 完成；要么在第一个子 `Observable` 完成之后，再去订阅第二个子 `Observable`。这样的行为其实已经在 `concatMap()` 中实现了（参见 3.1.5 节）。但是，`flatMap()` 会立即订阅所有的子流并将它们合并到一起，子流发布的任何事件都会被推送至下游。`flatMap()` 返回的所有子序列都会被合并，并且被平等对待，也就是说，RxJava 立即订阅所有的子流并将事件均匀地推送到下游之中。

```
Mon-0, Sun-0, Mon-1, Sun-1, Mon-2, Mon-3, Sun-2, Mon-4, Sun-3, Sun-4
```

如果你仔细跟踪所有延迟，会发现这个顺序实际是正确的。例如，虽然星期日（Sunday）是上游 `Observable` 中的第一个事件，但 `Mon-0` 事件是第一个出现的，因为星期一（Monday）生成的子流发布速度会更快。这也是 `Mon-4` 出现在 `Sun-3` 和 `Sun-4` 之前的原因。

3.1.5 使用concatMap()保证顺序

如果你真的想要保持下游事件的顺序，使其与上游事件的顺序完全契合，又该如何实现呢？换句话说，上游事件 N 产生的下游事件必须发生在 N+1 产生的事件之前。这里就涉及一个便捷的 concatMap 操作符，它和 flatMap() 语法完全一样，运行方式却非常不同，如下所示。

```
Observable
        .just(DayOfWeek.SUNDAY, DayOfWeek.MONDAY)
        .concatMap(this::loadRecordsFor);
```

现在的输出完全符合预期。

```
Sun-0, Sun-1, Sun-2, Sun-3, Sun-4, Mon-0, Mon-1, Mon-2, Mon-3, Mon-4
```

那么内部发生了什么呢？第一个事件（Sunday）从上游出现的时候，concatMap() 会订阅 loadRecordsFor() 产生的 Observable，并将产生的所有事件传递到下游。这个内部流完成时，concatMap() 会等待下一个上游事件（Monday）并重复以上过程。concatMap() 不会涉及任何的并发性，但是它保证了上游事件的顺序，避免出现重叠。

flatMap() 内部使用了 merge() 操作符，同时订阅所有的子 Observable，对它们不做任何的区分（参见 3.2.1 节）。这也是下游事件互相交叉的原因。但是，concatMap() 可以在技术上使用 concat() 操作符（参见 3.4.1 节）。concat() 只会先订阅第一个底层的 Observable，只有第一个完成之后，才会订阅第二个。

控制flatMap()的并发性

假设你有大量用户的一个列表，它们被包装在 Observable 中。每个 User 有一个 loadProfile() 方法，该方法会通过 HTTP 请求返回一个 Observable<Profile> 实例。我们的目标是尽快获取所有用户概况（profile），flatMap() 就是为了实现该目标而设计的，可以对上游的值进行并发计算，如下所示。

```
class User {
    Observable<Profile> loadProfile() {
        //发送HTTP请求……
    }
}

class Profile {/* ... */}

//...

List<User> veryLargeList = //...
Observable<Profile> profiles = Observable
        .from(veryLargeList)
        .flatMap(User::loadProfile);
```

乍看上去这种方式非常不错。Observable<User> 是从一个使用 from() 操作符的固定 List 生成的。因此，订阅它的时候，会将所有的用户立即释放出来。对于每个新 User，

flatMap() 都会调用 loadProfile() 并返回 Observable<Profile>。然后，flatMap() 透明地订阅每个新的 Observable<Profile>，将所有的 Profile 事件转发至下游。订阅内部 Observable<Profile> 就相当于发起新的 HTTP 连接。因此，假设我们有 10 000 个用户，那就会突然发起 10 000 个并发的 HTTP 请求。如果所有的这些请求都访问相同的服务器，预计得到的情况无外乎如下几种。

- 拒绝连接。
- 长时间等待和超时。
- 服务器停机。
- 遇到限速或者被加入黑名单。
- 整体的延迟增加。
- 客户端的问题，包括太多打开（open）状态的 Socket、线程，以及过多的内存消耗。

增加并发会在一定的程度上得到回报，但如果你尝试运行太多并发操作，最终将会导致大量的上下文切换、过高的内存和 CPU 占用，以及整体性能的下降。有种方案是在一定程度上减缓 Observable<User>，这样的话，它就不会一次性发布所有的 User。但是，调节这个延迟以实现最佳的并发级别是非常麻烦的。相反，flatMap() 有一个非常简单的重载形式，能够限制内部流的并发订阅总数。

```
flatMap(User::loadProfile, 10);
```

参数 maxConcurrent 限制了内部 Observable 的订阅数量。在实践中，flatMap() 接收前 10 个 User 时，它会为每个 User 调用 loadProfile()，但是来自上游的第 11 个 User 出现时，[3]flatMap() 不会再调用 loadProfile()。相反，它会等正在运行的内部流完成。因此，maxConcurrent 参数限制了 flatMap() 生成的后台任务的数量。

你可能已经发现，concatMap(f) 在语义上是与 flatMap(f, 1)（也就是 maxConcurrent 值为 1 的 flatMap()）等价的。其实，本来还可以多用几页的篇幅介绍 flatMap()，但是接下来还有更有意思的操作符。

3.2 多个Observable

转换单个 Observable 是非常有意思的，但是如果有更多的 Observable 需要协作又该怎么办呢？如果你了解 Java 中传统的并发编程，就会知道代码会充斥着 Thread 和 Executor，也会明白共享可变状态和同步是多么困难。幸而，在这样的环境中，RxJava 运行得会更好。同时，该库还有一个统一的方式来处理错误，涉及多个流的所有操作符均能使用这种方式。如果上游的任意一个源发布错误通知，它将会被传递到下游，并且以一个错误的形式完成下游的序列。如果多个上游的 Observable 都发生错误，第一个错误将会优先得到处理，其他的错误则会被舍弃（每个 Observable 只能发布一次 onError，参见 2.1 节）。最后，如果你想要继续进行处理，并且等所有正常事件都得到处理之后再去发布错误，很多操作符可以提供 *DelayError 的变种形式。

注 3：事实上，flatMap() 不能再接收 User 了，这一特性将在 6.2.4 节中详细探讨。

3.2.1　使用merge()将多个Observable合并为一个

还记得 3.1.2 节中的 `Observable<LicensePlate> recognize(CarPhoto photo)` 方法吗？这个方法会异步地根据 `CarPhoto` 识别 `LicensePlate`。我们曾经简单地提过这样的流可能会同时使用多个算法，有的可能会更快，有的可能会更精确。但是，我们并不想将这些细节暴露给外部世界，只是想有一个包含渐进式优化结果的流，这个结果来源于各个算法，从最快的到最精确的。

假设现在有三个支持 RxJava 的算法，每个算法都使用 `Observable` 进行了很好的封装。当然，每个算法可能会生成零个到无穷个结果。

```
Observable<LicensePlate> fastAlgo(CarPhoto photo) {
    //速度快但质量差
}

Observable<LicensePlate> preciseAlgo(CarPhoto photo) {
    //精确但代价高昂
}

Observable<LicensePlate> experimentalAlgo(CarPhoto photo) {
    //不可预测，但是可运行
}
```

我们想要达到的效果是同时运行这三个算法（参见 4.9.2 节，了解 RxJava 处理并发的更多细节）并尽快接收结果。我们并不关心是哪个算法发布了事件，只想要捕获所有的事件并将它们聚集到一个流中。这就是 `merge()` 操作符能实现的功能，如下所示。

```
Observable<LicensePlate> all = Observable.merge(
        preciseAlgo(photo),
        fastAlgo(photo),
        experimentalAlgo(photo)
);
```

以上代码有意将 `preciseAlgo()`（应该是最慢的一个）放到了第一个的位置上，以强调传递给 `merge()` 的 `Observable` 顺序是任意的。`merge()` 操作符将会保留一个对所有底层 `Observable` 的引用，有人订阅 `Observable<LicensePlate> all` 时，它会一次性地订阅所有的上游 `Observable`。不管是哪个 `Observable` 首先发布事件，这个事件都将会转发给 `all` 的 `Observer`。当然，`merge()` 遵循 Rx 的契约（参见 2.1 节），即便底层的流同时发布值，它也能保证事件是序列化的（不会出现重叠）。图 3-5 的弹珠图阐述了 `merge()` 是如何运行的。

`merge()` 操作符广泛应用于将多个相同类型的事件源视为统一来源的情景。[4] 同样，如果你想要进行 `merge()` 的 `Observable` 只有两个，那么可以使用 `obs1.mergeWith(obs2)` 实例方法。

需要注意，任何底层 `Observable` 出现的错误都会立即传递到 `Observer`。可以使用 `merge()` 的 `mergeDelayError()` 变种形式来推迟错误，这样直到其他的流都完成，错误通知才会发布。`mergeDelayError()` 甚至能够保证收集所有的异常，而不仅仅是第一个，并将它们封装到 `rx.exceptions.CompositeException`。

注 4：这是某些运算类型的联结（join）阶段。

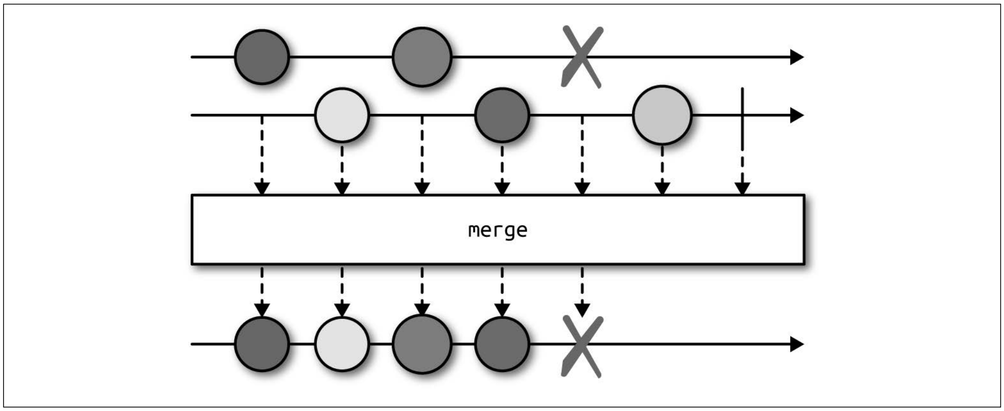


图 3-5

3.2.2 使用zip()和zipWith()进行成对地组合

压缩（zipping）指的是将两个（或更多）流组合起来的操作，在这个过程中，某个流中的每个事件必须要与其他流对应的事件进行成对组合。下游事件是通过组合每个流中的第一个事件，然后再组合第二个事件，以此类推生成的。因此，当所有的上游源都发布事件时，才会出现事件。如果想以某种形式组合多个彼此相关的流的结果，这个操作符是很有用的。或者与之相反，两个独立的流各自发布值，但是只有将它们组合在一起才有业务含义。图 3-6 的弹珠图阐述了它是如何运行的。

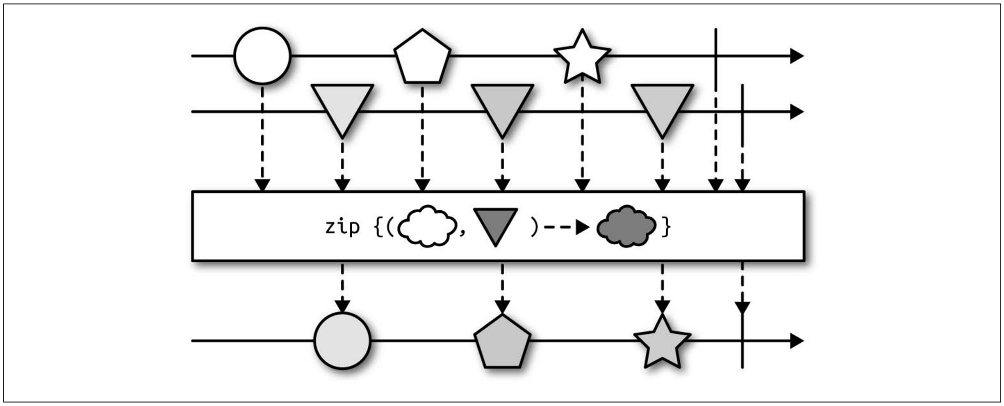


图 3-6

`zip()` 和 `zipWith()` 操作符是等价的。如果想要流畅地将一个流与另一个流进行组合，就可以使用后者，比如 `s1.zipWith(s2, ...)`。但是，如果想要组合的流超过了两个，那么可以使用 `Observable` 静态的 `zip()`，它最多接收 9 个流。

```
Observable.zip(s1, s2, s3...)
```

很多其他的操作符都有实例方法和静态方法的变种，比如 merge() 和 mergeWith()。为了理解 zip()，假设有两个独立的流，而这两个流是完全同步的。例如，WeatherStation 的 API 每分钟都要同时精准地发布温度和风力信息，如下所示。

```
interface WeatherStation {
    Observable<Temperature> temperature();
    Observable<Wind> wind();
}
```

样例必须要假设这两个 Observable 中的事件是同时发布的，也就是具有相同的频率。基于这个限制，可以很安全地将事件成对组合在一起，从而实现这两个流的合并。这意味着，某个流上出现事件时，必须将其临时持有，直到另一个事件出现，反之亦然。术语 zip 意味着要将两个流的事件联结在一起，一个来自左侧，一个来自右侧，循环重复。但是，在更通用的版本中，zip() 最多能够接收 9 个上游 Observable，并且只有它们全部发布事件的时候，才会形成下游的事件。

zip() 最合适的返回类型似乎是元组（tuple）或结对（pair，两个元素的元组）。但遗憾的是，Java 并没有针对结对的内置数据结构，而 RxJava 本身也没有任何的外部依赖。可以使用来自 Apache Commons Lang、Javaslang 或 Android SDK 的 Pair 实现。或者也可以提供组合成对事件的函数或数据结构，如下所示。

```
class Weather {
    public Weather(Temperature temperature, Wind wind) {
        //...
    }
}

//...

Observable<Temperature> temperatureMeasurements = station.temperature();
Observable<Wind> windMeasurements = station.wind();

temperatureMeasurements
    .zipWith(windMeasurements,
        (temperature, wind) -> new Weather(temperature, wind));
```

新的 Temperature 事件出现时，zipWith() 会等待 Wind（当然是没有阻塞的），反之亦然。这两个事件会传递到自定义的 lambda[5] 中，并组合成一个 Weather 对象。然后，循环往复该过程。zip() 是使用流的方式来进行描述的，甚至是无穷流。但是，你会发现为 Observable 使用 zipWith() 和 zip() 通常只会发布一个条目。这样的 Observable 一般是对某种请求或操作的异步响应。第 4 章将会讨论如何将 RxJava 用到真正的应用中。

现在学习一个样例，根据两个流中的所有值生成**笛卡儿积**。例如，可能有两个 Observable，一个代表棋盘的行（rank，从 1 到 8），另一个代表棋盘的列（file，从 a 到 h）。应该能够找到棋盘上所有 64 个可能的方格。

```
Observable<Integer> oneToEight = Observable.range(1, 8);
Observable<String> ranks = oneToEight
```

注 5：简短的 lambda 语法 Weather::new 在这里也适用。

```
        .map(Object::toString);
    Observable<String> files = oneToEight
        .map(x -> 'a' + x - 1)
        .map(ascii -> (char)ascii.intValue())
        .map(ch -> Character.toString(ch));

    Observable<String> squares = files
        .flatMap(file -> ranks.map(rank -> file + rank));
```

Observable 将会精确地发布 64 个事件：针对 a 它会生成 a1、a2...a8，然后是 b1、b2 等，直到最后达到 h7 和 h8。这是 flatMap() 另一个非常有意思的样例，每列（file）都会生成该列对应的所有可能的方格。现在，来看一个更现实的例子，它也会用到笛卡儿积。假设你想要规划在某个城市的一日游行程，但是只有天气晴朗并且有廉价航班和酒店的时候，该旅游行程才有效。为了实现这一点，需要将多个流联合在一起并形成所有可能出现的结果。

```
import java.time.LocalDate;

Observable<LocalDate> nextTenDays =
    Observable
        .range(1, 10)
        .map(i -> LocalDate.now().plusDays(i));

Observable<Vacation> possibleVacations = Observable
    .just(City.Warsaw, City.London, City.Paris)
    .flatMap(city -> nextTenDays.map(date -> new Vacation(city, date)))
    .flatMap(vacation ->
        Observable.zip(
            vacation.weather().filter(Weather::isSunny),
            vacation.cheapFlightFrom(City.NewYork),
            vacation.cheapHotel(),
            (w, f, h) -> vacation
        ));
```

Vacation 类如下所示。

```
class Vacation {
    private final City where;
    private final LocalDate when;

    Vacation(City where, LocalDate when) {
        this.where = where;
        this.when = when;
    }

    public Observable<Weather> weather() {
        //...
    }

    public Observable<Flight> cheapFlightFrom(City from) {
        //...
    }

    public Observable<Hotel> cheapHotel() {
        //...
```

```
    }
}
```

在上面的代码中发生了很多事情。首先，组合使用 range() 和 map() 生成从明天开始往后 10 天的日期。然后，使用 flatMap() 与三个城市进行组合，在这里我们不想使用 zip()，因为需要得到日期与城市所有可能的组合。对于每个组合，创建一个 Vacation 实例来封装它。现在到了真正的逻辑，使用 zip 操作符连接三个 Observable：Observable<Weather>、Observable<Flight> 和 Observable<Hotel>。根据指定的城市 / 日期是否有廉价的航班和酒店，后两个 Observable 将会返回零个或一个结果。尽管 Observable<Weather> 始终都会返回结果，但是我们使用 filter(Weather::sunny) 丢弃非晴朗的天气。最后会对这三个流进行 zip() 操作，每个流会发布零到一个条目。如果任意一个上游的 Observable 完成，zip() 就会立即完成并马上丢弃其他的流。由于这一特性，如果天气、航班或酒店中的任意一个 Observable 不存在，马上就会生成 zip() 的结果，这种情况下没有条目发布。这样形成了一个流，它由所有满足需求的行程组成。

不要惊讶于 zip 函数根本不考虑参数：(w, f, h) -> vacation。外层的 Vacation 流列出了每天所有可行的行程。但是，我们想要确保每个行程都出现天气、廉价航班和酒店信息。如果所有的条件都满足，就会返回一个 vacation 实例；否则，zip 根本就不会调用 lambda 表达式。

3.2.3 流之间不同步的情况：combineLatest()、withLatestFrom()和amb()

3.2.2 节做了一个非常大胆的假设，即假设两个 Observable 始终以相同的频率且在比较接近的时间点生成事件。但是，如果其中一个流的性能要比另一个流略好一些，那么较快 Observable 生成的事件需要花费越来越多的时间等待较慢流的事件。为了阐述这一点，首先对两个流进行 zip() 操作，这两个流按照完全相同的频率生成条目。

```
Observable<Long> red   = Observable.interval(10, TimeUnit.MILLISECONDS);
Observable<Long> green = Observable.interval(10, TimeUnit.MILLISECONDS);

Observable.zip(
    red.timestamp(),
    green.timestamp(),
    (r, g) -> r.getTimestampMillis() - g.getTimestampMillis()
).forEach(System.out::println);
```

red 和 green 这两个 Observable 按照相同的频率生成条目。为每个条目附加上 timestamp() 信息，这样就能知道发布的时间。

> timestamp()
>
> timestamp() 操作符使用 rx.schedulers.Timestamped<T> 类包装任意 T 类型的事件，这个类有两个属性：一个是原始的 T 类型的值，另一个是创建它时的 long 时间戳。

上面的 zip() 转换只是简单地对比了两个流中每个事件的创建时间。如果流是同步的，那么这个值大约等于零。但是，如果稍微降低其中一个 Observable 的速度，假设把 green 修改为 Observable.interval(11,MILLISECONDS)，情况将会大不相同。red 和 green 的时间差将会越来越大：red 能够被实时消费，但是它必须要等待。因为另外一个条目产生得更慢，这会增加消耗的时间。随着时间的推移，这种差异会累积起来，产生过期数据，甚至内存泄漏（参见 8.6 节）。在实践中，使用 zip() 必须要非常小心。

我们真正想要的效果是**任意**一个上游流产生事件时，就使用另外一个流最新的已知值。这就是 combineLatest() 能够发挥作用的地方了，如图 3-7 的弹珠图所示。

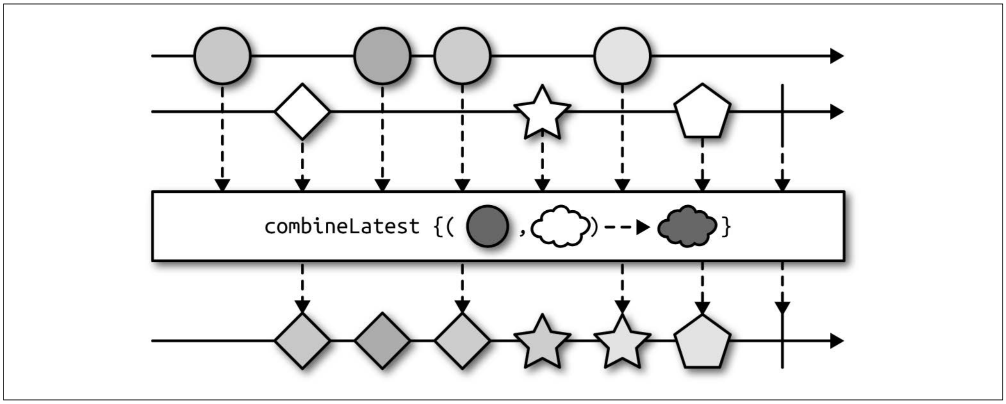


图 3-7

参考以下样例。一个流每隔 17 毫秒就会产生 S0、S1、S2...；而另一个流每隔 10 毫秒会产生 F0、F1、F2...（更快一些）。

```
import static java.util.concurrent.TimeUnit.MILLISECONDS;
import static rx.Observable.interval;

Observable.combineLatest(
    interval(17, MILLISECONDS).map(x -> "S" + x),
    interval(10, MILLISECONDS).map(x -> "F" + x),
    (s, f) -> f + ":" + s
).forEach(System.out::println);
```

将这两个流组合在一起，其中任意一个流生成事件时都会产生一个新的值。输出很快就会不同步了，但至少此时值能够实时被消费掉，较快的流不用再等待较慢的流。

```
F0:S0
F1:S0
F2:S0
F2:S1
F3:S1
F4:S1
F4:S2
F5:S2
F5:S3
...
```

```
F998:S586
F998:S587
F999:S587
F1000:S587
F1000:S588
F1001:S588
```

请注意，每个新 F 事件上的新条目是如何在下游中出现的：F0:S0、F1:S0 和 F2:S0。RxJava 在较快的流上发现了新事件，所以它就从较慢的流上得到最新的一个值（较快的流依然有两个事件在等待较慢的流上的一个事件），在这里也就是 S0，并生成一个新的值对。但是，具体哪个流是没有区别的：较慢流出现 S1 时，同样会取到较快流上最新的值（F2）并将其联合起来。大约 10 秒之后，我们就会看到 F1000:S588。所有的事件累加在一起：在这 10 秒的时间里，较快的流生成了大约 1000 个事件，而较慢的流只生成了 588 个（也就是用 10 秒除以 17 毫秒得到的结果）。

1. withLatestFrom()操作符

combineLatest 是对称的，也就是说它不会区分想要组合的子流。但在有些情况下，可以在某个流出现事件的时候，结合第二个流中的最新值生成一个事件，反之则不可以。换句话说，来自第二个流的事件并不会触发下游的事件，只有第一个流发布事件的时候才会用到它们。可以使用新的 withLatestFrom() 操作符来实现这一行为。下面使用相同的 slow 和 fast 流来阐述。

```
Observable<String> fast = interval(10, MILLISECONDS).map(x -> "F" + x);
Observable<String> slow = interval(17, MILLISECONDS).map(x -> "S" + x);
slow
    .withLatestFrom(fast, (s, f) -> s + ":" + f)
    .forEach(System.out::println);
```

在上面的样例中，slow 流是主导性的，slow 发布事件的时候，最终的 Observable **始终**都会发布一个事件，当然这样的前提是 fast 至少已经发布过一个事件了。与之相反，fast 流只是一个辅助者，只有 slow 发布事件时才会用到它。函数作为第二个参数传递给 withLatestFrom()，它会将 slow 流中的每个新值同 fast 流中的最新值结合在一起。但是，fast 流中的新值并不会传递至下游中。新的 slow 出现时，它们只会在内部更新。上述代码片段的输出显示所有的 slow 事件只会出现一次，而有些 fast 事件则会被丢弃。

```
S0:F1
S1:F2
S2:F4
S3:F5
S4:F7
S5:F9
S6:F11
...
```

在第一个 fast 事件出现之前的所有 slow 事件都会被悄悄丢弃，因为没有与它们联合的事件。按照设计，它就是这样运行的。如果你真的想要保留主导流中的所有事件，那么必须要确保其他的流要尽快发布一些虚拟的事件。例如，你可以让流预先发布一些虚拟的事件。下面的样例人为地延缓了 fast 流，使它的所有事件延迟了 100 毫秒（参见 3.1.3 节）。

如果没有虚拟事件，则将会丢失好几个 slow 中的事件。但是使用 startWith() 操作符，就可以创建一个衍生自 fast 的新 Observable。它马上就会有一个 FX 值，然后是原始 fast 流中的值。

```
Observable<String> fast = interval(10, MILLISECONDS)
        .map(x -> "F" + x)
        .delay(100, MILLISECONDS)
        .startWith("FX");
Observable<String> slow = interval(17, MILLISECONDS).map(x -> "S" + x);
slow
        .withLatestFrom(fast, (s, f) -> s + ":" + f)
        .forEach(System.out::println);
```

输出显示这里没有丢弃任何 slow 事件。但是，在开始的时候，多次看到了虚拟的 FX 事件，直到 100 毫秒之后第一个 F0 出现。

```
S0:FX
S1:FX
S2:FX
S3:FX
S4:FX
S5:FX
S6:F1
S7:F3
S8:F4
S9:F6
...
```

总的来说，startWith() 会返回一个新的 Observable，它在订阅的时候，马上就会发布一些常量值（如 "FX"），然后才是原始的 Observable。例如，下面的代码片段会依次生成 0、1 和 2。

```
Observable
    .just(1, 2)
    .startWith(0)
    .subscribe(System.out::println);
```

请参见 3.4 节，了解与之类似的 concat() 操作符的样例。

2. amb()操作符

最后一个有用的操作符是 amb()（以及 ambWith()），它会订阅上游其所操控的所有 Observable 并等待第一个事件的发布。其中有一个 Observable 发布第一个事件之后，amb() 会丢弃所有其他的流，接下来只跟踪第一个发布事件的 Observable，如图 3-8 的弹珠图所示。

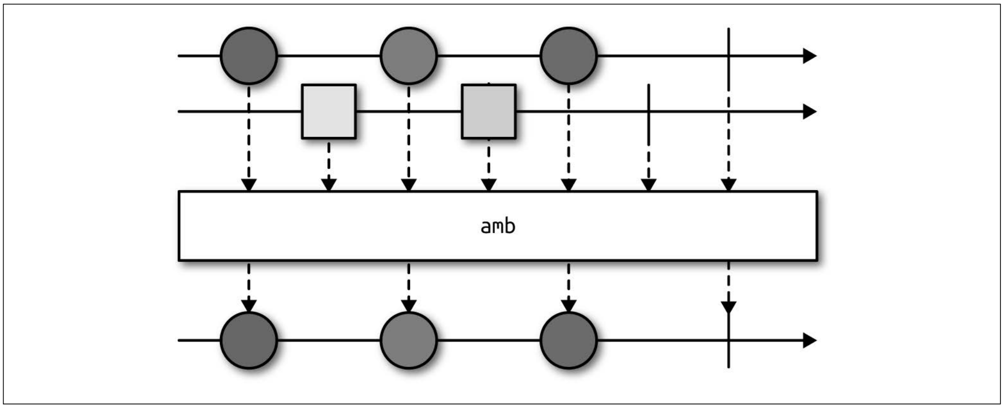


图 3-8

下面的样例阐述了针对两个流时，amb() 是如何运行的。请注意 initialDelay 参数，它决定了哪个 Observable 会首先发布事件。

```
Observable<String> stream(int initialDelay, int interval, String name) {
    return Observable
        .interval(initialDelay, interval, MILLISECONDS)
        .map(x -> name + x)
        .doOnSubscribe(() ->
            log.info("Subscribe to " + name))
        .doOnUnsubscribe(() ->
            log.info("Unsubscribe from " + name));
}

//...

Observable.amb(
        stream(100, 17, "S"),
        stream(200, 10, "F")
).subscribe(log::info);
```

你可以使用非静态的 ambWith() 编写相同功能程序的代码，但是它的易读性稍差一些，因为这样会隐藏掉 amb() 的对称性。下面的代码看起来像是基于第一个流使用第二个流，但实际上它们两者是被平等对待的。

```
stream(100, 17, "S")
        .ambWith(stream(200, 10, "F"))
        .subscribe(log::info);
```

不管你更喜欢哪个版本，它们都会生成相同的结果。slow 流产生事件的频率更低，它的第一个事件会在大约 100 毫秒之后出现，而 fast 流产生的第一个事件则会在 200 毫秒之后出现。amb() 做的事情就是先订阅这两个 Observable，它遇到 slow 流中的第一个事件之后，会立即取消对较快的流订阅，仅转发较慢流中的事件。

```
14:46:13.334: Subscribe to S
14:46:13.341: Subscribe to F
```

```
14:46:13.439: Unsubscribe from F
14:46:13.442: S0
14:46:13.456: S1
14:46:13.473: S2
14:46:13.490: S3
14:46:13.507: S4
14:46:13.525: S5
```

在进行调试的时候，`doOnSubscribe()` 和 `doOnUnsubscribe()` 是非常有用的（参见 7.4.1 节）。在订阅了 `S` 大约 100 毫秒之后，请注意 `amb()` 是如何取消订阅 `F` 的，这恰好是 `S` `Observable` 发布第一个事件的时间。此时，监听来自 `F` 的事件就没有任何意义了。

3.3 高级操作符：collect()、reduce()、scan()、distinct()和groupBy()

有些操作符能够进行更高级的转换操作，比如扫描整个序列并在这个过程中聚合一些值，如计算平均值。有些操作符甚至是有状态的，它们在序列往前推进的过程中管理内部的状态。这也是 `distinct` 的运行方式——缓存并丢弃访问过的值。

3.3.1 使用Scan和Reduce扫描整个序列

目前为止介绍的所有操作符都是基于单个事件（如过滤、映射或压缩）的。但有时候你希望聚合事件，从而对原始的流进行缩减或简化。以一个监控数据传输进度的 `Observable<Long>` 为例，每次数据发送成功，都会出现一个 `Long` 值，它代表了数据块的大小。这个信息有一定的用处，但是真正值得关心的是总计传输了多少字节。一个非常糟糕的主意就是使用全局变量，并在操作符内部对该变量进行修改。

```
import java.util.concurrent.atomic.LongAdder;


//有问题!
Observable<Long> progress = transferFile();

LongAdder total = new LongAdder();
progress.subscribe(total::add);
```

就像其他的共享状态一样，上述代码可能会导致令人非常恼火的并发缺陷。操作符中的 lambda 表达式可能会在任意的线程中执行，所以全局状态必须是线程安全的。另外，还必须要将延迟执行考虑进来。RxJava 通过可组合的操作符，可以尽量减少全局变量和可变性，但即便有 Rx 的保证，修改全局状态也是很容易出错的。除此之外，也不能再使用 Rx 操作符进一步地组合 `total` 值（比如阶段性地更新用户界面），传输完成时进行通知也会更加复杂。我们真正想要实现的是：新的数据块出现时，增量累积数据块的大小并报告当前总值。假设的流应该如下所示。

```
Observable<Long> progress =      //[10, 14, 12, 13, 14, 16]
Observable<Long> totalProgress = /* [10, 24, 36, 49, 63, 79]
```

```
    10
    10+14=24
          24+12=36
                36+13=49
                      49+14=63
                            63+16=79
*/
```

第一个条目会原样（as-is，10）传递至下游。但是，在第二个条目（14）传递到下游之前，它需要与上一个已发布的条目（10）相加，也就是要发布 24（前两个条目的和）。第三个条目（12）同样也会添加到结果流的上一个条目（24）上，也就是要发布 36。这个迭代式的过程会一直持续，直到上游的 Observable 完成。此时，最后发布的条目就是所有上游事件的总和。可以使用 scan() 操作符实现这个相对比较复杂的工作流。

```
Observable<Long> totalProgress = progress
    .scan((total, chunk) -> total + chunk);
```

scan() 会接收两个参数：上一次生成的值（也被称为**累加器**）以及上游 Observable 的当前值。在第一轮迭代中，total 就是来自 progress 的第一个条目；而在第二次迭代中，它变成了上一次 scan() 操作的结果值。如表 3-1 所示。

表3-1：一行代表了一次scan()循环

progress	total	chunk	totalProgress
10	-	-	-
14	10	14	24
12	24	12	36
13	36	13	49
14	49	14	63
16	63	16	79

scan() 就像一个推土机，它会遍历源（上游）Observable 并对所有条目进行累积。重载版本的 scan() 可以提供一个初始值（如果初始值与第一个元素的值不同，可以采用这种方式）。

```
Observable<BigInteger> factorials = Observable
    .range(2, 100)
    .scan(BigInteger.ONE, (big, cur) ->
        big.multiply(BigInteger.valueOf(cur)));
```

factorials 将会生成 1、2、6、24、120、720，等等。注意，上游的 Observable 是从 2 开始的，下游却是从 1 开始的，这是因为设置了初始值（BigInteger.ONE）。根据经验，最后的 Observable 应该总与累加器的类型相同。所以，如果你没有为累积器提供自定义的初始值，那么 scan() 返回的 T 类型应该不会发生变化的。否则（比如在 factorial 样例中），结果就会是 Observable<BigInteger> 类型的，因为 BigInteger 是初始值的类型。显然，在整个扫描过程中，这个类型是不能变化的。

有时候我们并不关心中间结果，只关心最终结果。例如，想要计算传输的总计字节数，而不是中间的进度，或者想要累积某个可变数据结构（如 ArrayList）中的所有值，每次都

会添加一个条目。reduce() 就是专门为此设计的。一个显而易见的警告：如果你的序列是无穷的，scan() 会持续为每个上游的事件发布事件，但是 reduce() 将不会发布任何的事件。假设有一个由 CashTransfer 对象组成的源，它有一个返回 BigDecimal 的 getAmount() 方法。想要计算累积的转账的总额，如下的两种转换方式是等价的，它们都会从零（ZERO）开始遍历所有的转账记录并累积总额。

```
Observable<CashTransfer> transfers = //...;


Observable<BigDecimal> total1 = transfers
    .reduce(BigDecimal.ZERO,
        (totalSoFar, transfer) ->
            totalSoFar.add(transfer.getAmount()));
Observable<BigDecimal> total2 = transfers
    .map(CashTransfer::getAmount)
    .reduce(BigDecimal.ZERO, BigDecimal::add);
```

两种转换会生成相同的结果，第二种方式尽管有两个步骤看起来却更简洁。这也是为何推荐使用更小、更具组合性的转换，而不是采用单个大型的转换。你可能也看出来了，reduce() 基本上只接收最后一个元素的 scan()。可以按照如下的方式实现。

```
public <R> Observable<R> reduce(
        R initialValue,
        Func2<R, T, R> accumulator) {
    return scan(initialValue, accumulator).takeLast(1);
}
```

如上所示，reduce() 其实就是扫描整个 Observable，但是丢弃了除最后一个元素之外的其他所有元素（参见 3.4 节）。

3.3.2　使用可变的累加器进行缩减：collect()

现在，让我们将类型为 T 的有限事件流转换为另一个流，这个被转换而成的流只有一个 List<T> 类型的事件。当然，这个事件只有上游 Observable<T> 完成的时候才会被发布出来。

```
Observable<List<Integer>> all = Observable
    .range(10, 20)
    .reduce(new ArrayList<>(), (list, item) -> {
        list.add(item);
        return list;
    });
```

reduce() 的这个样例会从一个空的 ArrayList<Integer>（累加器）开始，并且会将发布的每个条目添加到该 ArrayList 中。负责进行缩减（累加）的 lambda 表达式必须返回一个新版本的累加器。令人遗憾的是，List.add() 并不会返 List，相反，它返回的是一个 boolean 值。因此，这里需要一个显式的 return 语句。为了避免这种冗余，可以使用 collect() 操作符。它的原理与 reduce() 几乎相同，只不过假设针对每个事件会使用相同的可变累加器，而不是每次都返回一个不可变的新累加器（将其与不可变的 BigInteger 样例进行一下对比），如下所示。

```
Observable<List<Integer>> all = Observable
    .range(10, 20)
    .collect(ArrayList::new, List::add);
```

collect() 另外一个非常有用的用例就是将所有的事件累积到一个 StringBuilder 中。在这种情况下，累加器是一个空的 StringBuilder，操作就是将一个条目附加到该生成器。

```
Observable<String> str = Observable
        .range(1, 10)
        .collect(
                StringBuilder::new,
                (sb, x) -> sb.append(x).append(", "))
        .map(StringBuilder::toString);
```

与其他的 Observable 操作符类似，reduce() 和 collect() 都是非阻塞的。所以只有上游完成的时候，才会发布最后的 List<Integer>，它包含 Observable.range(10, 20) 发布的所有数字；异常则会正常传递。将 Observable<T> 转换为 Observable<List<T>> 非常常见，所以 RxJava 提供了内置的 toList() 操作符。参见 4.2 节中的现实世界用例。

3.3.3　使用single()断言的Observable只有一个条目

有些 Observable 按照定义能且仅能发布一个值。例如，上述的代码片段始终都只发布一个 List<Integer>，即便这个列表有可能是空的。在这种情况下，使用 single() 操作符就是很有价值的了。它不会以任何形式改变上游的 Observable，然而，它确保该 Observable 能且仅能发布一个事件。如果这种假设是错误的，将会收到一个异常，而不是难以预料的结果。

3.3.4　使用distinct()和distinctUntilChanged()丢弃重复条目

简单随机值组成的无穷流是非常有用的，特别是在与其他流组合的时候。如下的 Observable 会生成 0 到 1000 之间的伪随机 Integer 值。

```
Observable<Integer> randomInts = Observable.create(subscriber -> {
    Random random = new Random();
    while (!subscriber.isUnsubscribed()) {
        subscriber.onNext(random.nextInt(1000));
    }
});
```

显然，这里可能会出现重复的值，take(1001) 中肯定至少有一个重复的值。[6] 但是，如果我们想窥视一个更少的（比如 10 个）、**唯一的**随机值，又该怎么办呢？内置的 distinct() 操作符会自动丢弃上游 Observable 中出现过的事件，确保只有唯一的事件才会被传递到下游。

```
Observable<Integer> uniqueRandomInts = randomInts
        .distinct()
        .take(10);
```

注 6：仅有 1000 个可能唯一的 nextInt(1000) 结果。

每次有新的值从上游 Observable(randomInts) 发布出来，distinct() 操作符内部就会确保这个值没有出现过。这种比较是通过 equals() 和 hashCode() 来完成的，所以要确保这两种方法是按照 Java 指南实现的（两个相等的对象**必须**要具有相同的 hash code）。有意思的是，take(1001) 最终会以随机顺序发布 0 到 999 之间的所有单个值，但是永远不会完成，因为在 0 到 999 之间没有第 1001 个 int 类型的唯一值。

2.5 节介绍过 Observable<twitter4j.Status>，它会发布社交媒体网站 Twitter 的状态更新。每次用户一发布状态更新，这个 Observable 就会推送新的事件。Status 对象包含了多个属性，比如 getText()、getUser() 等。鉴于重复几乎不可能出现，所以 distinct() 操作符对于 Status 事件并没有太大的意义。但是，如果只想查看每个用户第一次更新的文本（status.getUser().getId() 会返回 long 类型的值），该怎样实现呢？显然，可以抽取这个唯一的属性并基于它运行 distinct()。

```
Observable<Status> tweets = //...

Observable<Long> distinctUserIds = tweets
        .map(status -> status.getUser().getId())
        .distinct();
```

遗憾的是，在执行 distinct() 的时候，原始的 Status 对象丢失了。真正想要的是这种方式：抽取事件的属性，从而决定唯一性。如果抽取出来的属性（也就是所谓的 key）之前已经出现过，那么两个事件就被认为是相等的（后者会被丢弃），如下所示。

```
Observable<Status> distinctUserIds = tweets
        .distinct(status -> status.getUser().getId());
```

key 返回的值会使用 equals() 和 hashCode() 与已有的 key 进行对比。需要注意的一点是，distinct() 必须要记住到目前为止出现的所有事件 /key（参见 3.6 节，想要针对唯一的事件仅处理一次的时候，distinct() 是非常有用的）。

在实践中，distinctUntilChanged() 通常更有用。在使用 distinctUntilChanged() 的时候，如果给定的事件与上一个事件相同，就会被丢弃（默认会使用 equals() 进行对比）。如果能够接收到某个测量数据的稳定流，并且想要在测量值发生变化的时候得到通知，那么 distinctUntilChanged() 是最适合的。3.2.2 节用到了 Observable<Weather>，其中的 Weather 有两个属性：Temperature 和 Wind。新的 Weather 可能每分钟就出现一次，但是天气的变化却没有那么频繁。所以重复的事件会被丢弃，我们只关注变更，如下所示。

```
Observable<Weather> measurements = //...

Observable<Weather> tempChanges = measurements
        .distinctUntilChanged(Weather::getTemperature);
```

上述的代码片段只有在温度发生变化（Wind 的变化不考虑在内）的时候，才会发布 Weather 事件。显然，如果想要在 Temperature 或 Wind 发生变化的时候发布事件，那么无参的 distinctUntilChanged() 就能很好地实现该功能，当然这里假设 Weather 实现了 equals() 方法。distinct() 和 distinctUntilChanged() 的重要区别在于后者可以产生重复的事件，但前提是重复的值被另一个不同值隔开。例如，每天都可能会出现相同的温度，但是它们会被更冷或更热的温度隔开。另外，distinctUntilChanged() 必须要记住上一个

出现的值。与之相对的是 distinct()，它必须从流的起始就跟踪所有唯一的值。这意味着，相对于 distinct()，distinctUntilChanged() 的内存消耗是可预测的固定值。

3.4 使用skip()、takeWhile()等进行切片和切块

从来都没有必要阅读整个流，在处理 hot 类型的无穷流时更应如此，当然这一特性并不局限于这种类型的流。实际上，对 Observable 进行切片，并且只使用其中很小的一个子集是非常常见的。本节中的大多数操作符都有样例，除非它们本身符合最小惊讶原则。而像 take() 或 last() 这样操作符是非常有用的，不能省略。如下是这些操作符的一个非详尽列表。

- take(n) 和 skip(n)

  take(n) 操作符会在上游发布完前 n 个事件之后提前截断源 Observable，随后取消订阅（如果上游的条目达不到 n 个，那么就会提前完成了）。skip(n) 则恰好相反，它会丢弃前 n 个元素，然后把上游 Observable 的第 n+1 个事件作为起点，开始发布事件。这两个操作符都具有一定的容错性：负数会被视为零，超出 Observable 的大小也不会被视为缺陷。

```
Observable.range(1, 5).take(3); //[1, 2, 3]
Observable.range(1, 5).skip(3); //[4, 5]
Observable.range(1, 5).skip(5); //[]
```

- takeLast(n) 和 skipLast(n)

  这是另外一对具有自描述性的操作符。takeLast(n) 只会发布流在完成之前的最后 n 个值。在内部，这个操作符必须要有一个缓冲区，保存最后的 n 个值，接收到完成通知后，它会立即发布整个缓冲区的内容。在无穷流上调用 takeLast() 是没有意义的，因为它不会发布任何的内容——流永远不会终止，所以也就不会有**最后的事件**。而 skipLast(n) 则会发布上游 Observable 中除了最后 n 个事件之外的其他所有事件。在内部，skipLast() 接收到 n+1 个事件的时候，就可以发布第一个值，在接收到第 n+2 个事件时，就可以发布第二个值，以此类推。

```
Observable.range(1, 5).takeLast(2); //[4, 5]
Observable.range(1, 5).skipLast(2); //[1, 2, 3]
```

- first() 和 last()

  无参的 first() 和 last() 操作符可以分别通过 take(1).single() 和 takeLast(1).single() 来实现，这种方式很好地描述了两个操作符的行为。额外的 single() 操作符会确保下游的 Observable 只会发布一个值或者异常。另外，first() 和 last() 都有接收断言的重载版本来实现。如果有断言，它们返回的就不一定真的是第一个 / 最后一个值，而是满足给定条件的第一个 / 最后一个值。

- takeFirst(predicate)

  takeFirst(predicate) 操作符可以表述为 filter(predicate).take(1)。这个操作符与 first(predicate) 的唯一区别在于如果没有匹配的值，它不会抛出 NoSuchElementException。

- takeUntil(predicate) 和 takeWhile(predicate)

  takeUntil(predicate) 和 takeWhile(predicate) 的关联性很强。takeUntil() 会发布上

游 Observable 中的值，但是在发布完第一个**匹配** predicate 的值之后，它就会完成并取消订阅。takeWhile(predicate) 则与之相反，只要这些值与给定的断言匹配，它会一直发布值。所以，区别在于 takeUntil() 会发布第一个不匹配的值，而 takeWhile() 则不会。这些操作符非常重要，它们基于发布的事件，有条件地对 Observable 取消订阅。否则，操作符就需要以某种方式与 Subscription 实例进行交互了（参见 2.3 节），操作符在调用的时候，这个实例其实是无法得到的。

```
Observable.range(1, 5).takeUntil(x -> x == 3); //[1, 2, 3]
Observable.range(1, 5).takeWhile(x -> x != 3); //[1, 2]
```

- elementAt(n)

  根据索引抽取特定条目的需求并不常见，针对这种情况可以使用内置的 elementAt(n) 操作符。它非常严格，如果上游 Observable 的长度不够或者索引为负时，它会产生 IndexOutOfBoundsException。当然，它返回的是与上游类型 T 相同的 Observable<T>。

- ...OrDefault() 操作符

  这一节中的很多操作符都很严格，可能会导致异常的抛出，比如上游 Observable 为空时，执行 first() 操作就会抛出异常。在这种情况下，有很多的 ...OrDefault() 操作符能够将异常替换为一个默认值。这些操作符的作用不言自明：elementAtOrDefault()、firstOrDefault()、lastOrDefault() 以及 singleOrDefault()。

- count()

  count() 是一个非常有趣的操作符，它会计算上游 Observable 发布了多少个事件。顺便提一下，如果你需要知道上游发布的条目中有多少满足给定的断言，那么按照习惯用法，filter(predicate).count() 能够实现该功能。所有的操作符都是延迟执行的，所以对于非常大型的流，这种方式依然可行。显然，对于无穷流来说，count() 并不会发布任何的值。你可以使用 reduce() 很容易地实现 count()，如下所示。

```
Observable<Integer> size = Observable
        .just('A', 'B', 'C', 'D')
        .reduce(0, (sizeSoFar, ch) -> sizeSoFar + 1);
```

- all(predicate)、exists(predicate) 和 contains(value)

  有时候，确保给定 Observable 的所有事件均匹配某个断言是非常有用的。如果上游的 Observable 正常完成，并且所有的值均匹配断言，那么 all(predicate) 会发布 true。一旦发现第一个不符合断言的值，它就会发布 false。exists(predicate) 与 all() 完全相反，发现第一个匹配断言的值之后，它就会发布 true；但是如果上游的 Observable 正常完成，并且没有发现任何匹配的值，那么它就会发布 false。通常，在 exists() 中的断言会与某些常量进行对比，在这种情况下，你还可以使用 contains() 操作符。

```
Observable<Integer> numbers = Observable.range(1, 5);

numbers.all(x -> x != 4);    //[false]
numbers.exists(x -> x == 4); //[true]
numbers.contains(4);         //[true]
```

3.4.1 组合流的方式：concat()、merge()和switchOnNext()

concat()（以及实例方法 concatWith()）能够将两个 Observable 连接在一起：第一个 Observable 完成的时候，concat() 会订阅第二个。非常重要的一点，当且仅当第一个 Observable 完成的时候，concat() 才会订阅第二个 Observable（参见 3.1.5 节）。concat() 甚至能够将不同的操作符用到同一个上游 Observable 上。例如，如果你只想接收一个非常长的流的前几个和最后几个条目，那么可以通过如下的方式实现。

```
Observable<Data> veryLong = //...
final Observable<Data> ends = Observable.concat(
        veryLong.take(5),
        veryLong.takeLast(5)
);
```

需要注意，上面的代码订阅了 veryLong 两次，这可能并非想要的效果。concat() 的另外一个样例就是在第一个流不发布任何内容条目的情况下，提供备用（fallback）值。

```
Observable<Car> fromCache = loadFromCache();
Observable<Car> fromDb = loadFromDb();

Observable<Car> found = Observable
        .concat(fromCache, fromDb)
        .first();
```

Observable 是延迟执行的，所以 loadFromCache() 和 loadFromDb() 实际上都还没有加载数据。如果缓存为空，loadFromCache() 可能不发布任何值就完成了，但是 loadFromDb() 始终都能返回一个 Car。concat() 之后的 first() 会首先订阅 fromCache，如果它能发布一个条目出来，样例将不再订阅 fromDb。但是，如果 fromCache 为空，concat() 会继续订阅 fromDb 并从数据库中加载数据。

实际上，concat() 操作符与 merge() 和 switchMap() 的关系非常密切。concat() 的运行方式类似于普通 List<T> 的连接：首先，它从第一个流中提取所有的条目，并且仅在第一个流完成的情况下，才会开始消费第二个流中的条目。当然，与之前看到的所有操作符一样，concat() 是非阻塞的，只有底层的流发布事件的时候，它才会随之发布事件。现在，对比一下 concat() 和 merge()（参见 3.2.1 节），以及马上要介绍的 switchOnNext()。

假设有一组人，每个人都有一个麦克风。每个麦克风都建模为 Observable<String>，其中每个事件代表一个单词。显然，事件在说出这些单词的时候才能出现。为了模拟这种行为，我们会创建一个简单的 Observable 进行演示，它本身是非常有意思的。如下所示。

```
Observable<String> speak(String quote, long millisPerChar) {
    String[] tokens = quote.replaceAll("[:,]", "").split(" ");
    Observable<String> words = Observable.from(tokens);
    Observable<Long> absoluteDelay = words
        .map(String::length)
        .map(len -> len * millisPerChar)
        .scan((total, current) -> total + current);
    return words
        .zipWith(absoluteDelay.startWith(0L), Pair::of)
        .flatMap(pair -> just(pair.getLeft())
```

```
            .delay(pair.getRight(), MILLISECONDS));
}
```

上述的代码片段非常复杂，所以先逐行学习一下。以 String 的形式接收一个任意的文本并将其拆分为单词，使用正则表达式删除掉标点符号。现在，对于每个单词，计算说出该单词需要的时间，用单词的长度乘以 millisPerChar 就可以获得。然后，根据时间的推移将这些单词传递出去，这样的话，每个单词都会按照上述样例计算出来的延迟依次出现。显然，简单的 from 操作符是不够的。

```
Observable<String> words = Observable.from(tokens);
```

我们想让单词按照一定的延迟出现，这个延迟要基于上一个单词的长度。第一个比较原始的实现方式就是直接根据每个单词的长度决定延迟。

```
words.flatMap(word -> Observable
    .just(word)
    .delay(word.length() * millisPerChar, MILLISECONDS));
```

这种方式是不正确的。Observable 首先会同时发布出所有单字母单词。然后，稍等片刻，它会发布出所有含有两个字母的单词，随后是三个字母的单词。但是，我们想要实现的效果是立即发布第一个单词，然后基于一定的延迟发布第二个单词，这里的延迟要由第一个单词的长度决定。这听起来非常复杂，但其实非常有意思。首先，根据 words 构建一个辅助流，它只包含每个单词引发的的相对延迟，如下所示。

```
words
    .map(String::length)
    .map(len -> len * millisPerChar);
```

假设 millisPerChar 的值为 100，words 的内容是 Though this be madness，首先得到了如下的流：600，400，200，700。如果只是按照持续时间来对每个单词进行 delay() 操作，be 会是第一个出现的单词，其他单词的顺序也会被打乱。但真正想要的是绝对延迟的累加值，如：600，600 + 400 = 1000，1000 + 200 = 1200，1200 + 700 = 1900。通过 scan() 操作符，这非常容易实现（参见 3.3.1 节）。

```
Observable<Long> absoluteDelay = words
    .map(String::length)
    .map(len -> len * millisPerChar)
    .scan((total, current) -> total + current);
```

现在，有了一个单词的序列和一个由每个单词绝对延迟组成的序列，我们可以将这两个流联合起来。这就是 zip() 能够发挥作用的地方，如下所示。

```
words
    .zipWith(absoluteDelay.startWith(0L), Pair::of)
    .flatMap(pair -> just(pair.getLeft()))
```

这样做很有意义，因为知道这两个流具有完全相同的大小，并且彼此完全同步，或者可以说几乎同步。我们不想让第一个单词有任何的延迟，但是，第一个单词的长度会影响第二个单词的延迟，第一个单词和第二个单词的总长度则会影响第三个单词的延迟，以此类推。通过为 absoluteDelay 添加一个初始值为 0 的值，能够很容易地实现这一点。

```
import org.apache.commons.lang3.tuple.Pair;

words
    .zipWith(absoluteDelay.startWith(0L), Pair::of)
    .flatMap(pair -> just(pair.getLeft())
        .delay(pair.getRight(), MILLISECONDS));
```

为单词构建一个与之配对的序列，也就是每个单词的绝对延迟，并且确保第一个单词是没有延迟的。这些配对的值如下所示。

```
(Though, 0)
(this, 600)
(be, 1000)
(madness, 1200)
...
```

这是说话的时间线，也就是每个单词以及与之相关的时间点。需要做的其余部分就是将每对元素转换成随时间不断出现的单元素 `Observable`。

```
flatMap(pair -> just(pair.getLeft())
    .delay(pair.getRight(), MILLISECONDS));
```

在完成这些准备工作之后，我们可以看一下 `concat()`、`merge()` 和 `switchOnNext()` 三者的区别。假设有三个人正在引述威廉·莎士比亚的《哈姆雷特》，如下所示。

```
Observable<String> alice = speak(
        "To be, or not to be: that is the question", 110);
Observable<String> bob = speak(
        "Though this be madness, yet there is method in't", 90);
Observable<String> jane = speak(
        "There are more things in Heaven and Earth, " +
        "Horatio, than are dreamt of in your philosophy", 100);
```

你可以看到，每个人的节奏都有轻微的不同，这是 `millisPerChar` 定义的。如果所有人同时说话，会怎么样呢？ RxJava 能够回答这个问题。

```
Observable
    .merge(
        alice.map(w -> "Alice: " + w),
        bob.map(w   -> "Bob:   " + w),
        jane.map(w  -> "Jane:  " + w)
    )
.subscribe(System.out::println);
```

输出非常混乱，每个人说出的单词都交织在一起，我们听到的都是噪声。如果不是每个语句都有个前缀，以下的输出内容会非常难以理解。

```
Alice: To
Bob:   Though
Jane:  There
Alice: be
Alice: or
Jane:  are
Alice: not
```

```
Bob:   this
Jane:  more
Alice: to
Jane:  things
Alice: be
Bob:   be
Alice: that
Bob:   madness
Jane:  in
Alice: is
Jane:  Heaven
Alice: the
Bob:   yet
Alice: question
Jane:  and
Bob:   there
Jane:  Earth
Bob:   is
Jane:  Horatio
Bob:   method
Jane:  than
Bob:   in't
Jane:  are
Jane:  dreamt
Jane:  of
Jane:  in
Jane:  your
Jane:  philosophy
```

这就是 `merge()` 的运行方式：它立即订阅每个人对应的单词并将其转发至下游，而不管过程中这些单词是哪个人说的。如果两个流几乎同时发布事件，那么它们都会立即被转发至下游。在这个操作符中没有事件的缓冲和暂停。

如果将 `merge()` 替换为 `concat()` 操作符，情况会发生很大的变化。

```
Alice: To
Alice: be
Alice: or
Alice: not
Alice: to
Alice: be
Alice: that
Alice: is
Alice: the
Alice: question
Bob:   Though
Bob:   this
Bob:   be
Bob:   madness
Bob:   yet
Bob:   there
Bob:   is
Bob:   method
Bob:   in't
```

```
Jane:  There
Jane:  are
Jane:  more
Jane:  things
Jane:  in
Jane:  Heaven
Jane:  and
Jane:  Earth
Jane:  Horatio
Jane:  than
Jane:  are
Jane:  dreamt
Jane:  of
Jane:  in
Jane:  your
Jane:  philosophy
```

现在，顺序完全没有问题了。concat(alice, bob, jane) 首先会订阅 alice，并持续转发这个 Observable 的事件，直到它的内容耗尽并完成。然后，concat() 会切换到 bob 上。请思考一下，hot 类型和 cold 类型的 Observable 的差异。在使用 merge() 的时候，所有流的全部事件都会进行转发，因为 merge() 会立即订阅事件流。但是，concat() 只会订阅第一个流，所以，如果是 hot 类型的 Observable，你可能会得到不同于预期的输出。第一个 Observable 完成的时候，第二个 Observable 可能会发布完全不同的事件序列。需要注意的是，concat() 并不会缓冲第二个 Observable 的事件，直到第一个 Observable 完成，它只是延迟对第二个 Observable 的订阅。

switchOnNext() 是以完全不同的方式进行组合的操作符。假设有一个 Observable<Observable<T>>，这个流中的每个事件本身也是一个流。这种情况是非常有意义的。例如，如果你有一组移动电话会不断连接和断开网络（外部流），每个新的连接都是一个独立的事件，但是每个这样的事件都是由独立心跳信息（Observable<Ping>）组成的流。在这样的场景下，会有一个 Observable<Observable<String>>，其中每个内部流代表每个人（即 alice、bob 或 jane）的引述。

```
import java.util.Random;

Random rnd = new Random();
Observable<Observable<String>> quotes = just(
                alice.map(w -> "Alice: " + w),
                bob.map(w   -> "Bob:   " + w),
                jane.map(w  -> "Jane:  " + w));
```

首先，将 alice、bob 和 jane 的 Observable 包装到一个 Observable<Observable<String>> 中。重申一下：quotes Observable 立即发布三个事件，每个事件都是一个内部 Observable<String>。每个内部的 Observable<String> 代表每人说的单词。为了阐述 switchOnNext() 是如何运行的，应该延迟内部 Observable 的发布。以下代码并不是延迟这个 Observable（变种 A）中的每个单词，而是延迟整个 Observable（变种 B 略有差异）。

```
//A
map(innerObs ->
        innerObs.delay(rnd.nextInt(5), SECONDS))
```

```
//B
flatMap(innerObs -> just(innerObs)
        .delay(rnd.nextInt(5), SECONDS))
```

在变种 *A* 中，Observable 会立即出现在外层流中，但是在一定延迟之后才会发布事件。在变种 *B* 中，我们及时转移了整个 Observable 的事件，所以它出现在外层 Observable 中的时间要晚得多。现在，已经介绍了为何需要这么复杂的操作符。静态的 concat() 和 merge() 操作符都能操作由固定列表组成的 Observable，或者由 Observable 组成的 Observable。而对 switchOnNext() 来说，后者会生效。

switchOnNext() 首先会订阅一个外层的 Observable<Observable<T>>，这个外层 Observable 会发布内层的 Observable<T>。第一个内层 Observable<T> 出现的时候，这个操作符就会订阅它并开始往下游推送 T 类型的事件。下一个内层 Observable<T> 出现的话，会发生什么呢？ switchOnNext() 将会取消对第一个 Observable<T> 的订阅，从而丢弃第一个 Observable<T>，然后切换至下一个 Observable<T>（这种做法也符合它的名称）。换句话说，如果有一个用流组成的流时，switchOnNext() 只会往下游转发最后的内部流的事件，即便旧的流依然在生成新的事件也会被丢弃。

《哈姆雷特》样例会如下所示。

```
Random rnd = new Random();
Observable<Observable<String>> quotes = just(
                alice.map(w -> "Alice: " + w),
                bob.map(w   -> "Bob:   " + w),
                jane.map(w  -> "Jane:  " + w))
        .flatMap(innerObs -> just(innerObs)
                .delay(rnd.nextInt(5), SECONDS));

Observable
        .switchOnNext(quotes)
        .subscribe(System.out::println);
```

因为随机性，这个样例可能产生的输出如下所示。

```
Jane:  There
Jane:  are
Jane:  more
Alice: To
Alice: be
Alice: or
Alice: not
Alice: to
Bob:   Though
Bob:   this
Bob:   be
Bob:   madness
Bob:   yet
Bob:   there
Bob:   is
Bob:   method
Bob:   in't
```

每个人在开始讲话之前，都会有 0 秒到 4 秒的任意延迟。在这次执行中，首先出现的是 Jane 的 Observable<String>，但是在讲完几个单词之后，Alice 的 Observable<String> 在外层 Observable 中出现了。此时，switchOnNext() 会取消对 jane 的订阅，然后我们就再也听不到她的引述了。这个 Observable 会被丢弃和忽略，switchOnNext() 此时只会监听 alice。但是，因为 Bob 的引述出现，这个内层 Observable 随后也会被中断。

从理论上讲，如果内层 Observable 不出现重叠，也就是在当前 Observable 完成之后，下一个 Observable 才出现，switchOnNext() 能够生成内层 Observable 的所有事件。

如果每个内层 Observable 只是延迟事件（还记得变种 *A* 的实现吗？），而不是延迟 Observable 本身，会怎样呢？这样，三个内层 Observable 会同时出现在外层 Observable 中，而 switchOnNext() 只会订阅其中之一。

3.4.2　使用groupBy()实现基于标准的切块流

与领域驱动设计（domain-driven design，关于领域驱动设计的更多信息，请参阅 Vaughn Vernon 的《实现领域驱动设计》）经常一起使用的一项技术叫作事件溯源（event sourcing）。在这种架构风格中，数据不会以当前状态快照的形式进行存储，也不能进行修改，也就是不能使用 SQL UPDATE 查询。相反，它会将已经发生的事情以不可变领域事件（fact）的形式保存在只能追加的数据存储中。使用这种设计永远不会覆盖任何数据，实际上就是免费得到了一个审计日志。除此之外，查看实时数据的唯一方式就是从一个空视图（empty view）开始，依次应用这些 fact。

在事件溯源中，基于初始的空状态应用事件被称为**投影**（projection）。[7] 一个 fact 源可以驱动多个投影。例如，可能会有与预定系统相关的 fact 流，比如 TicketReserved、ReservationConfirmed 和 TicketBought，这里的名称使用过去式是非常重要的，因为 fact 反映的都是发生过的操作和事件。从一个 fact 源（也是唯一的事实来源），可以衍生多个投影，如下所示。

- 所有确认预定的清单。
- 今天所有取消预定的清单。
- 每周的总收入。

系统演化的时候，可以丢弃旧的投影并构建新的，这主要是因为能够立即以 fact 的形式收集数据。假设想要构建一个包含所有预定信息及其状态的投影。为了实现这一点，必须要消费所有的 ReservationEvents，并将其应用到合适的预定信息中。针对不同类型的事件，每个 ReservationEvent 都有对应的子类，如 TicketBought。同时，每个事件都有一个预定信息的 UUID，用来判断要将事件应用到哪个预定信息上。

```
FactStore factStore = new CassandraFactStore();
Observable<ReservationEvent> facts = factStore.observe();
facts.subscribe(this::updateProjection);

//...
```

注 7：参见《实现领域驱动设计》附录 A 的“阅读模型投射”部分。

```
void updateProjection(ReservationEvent event) {
    UUID uuid = event.getReservationUuid();
    Reservation res = loadBy(uuid)
        .orElseGet(() -> new Reservation(uuid));
    res.consume(event);
    store(event.getUuid(), res);
}

private void store(UUID id, Reservation modified) {
    //...
}

Optional<Reservation> loadBy(UUID uuid) {
    //...
}

class Reservation {

    Reservation consume(ReservationEvent event) {
        //改变自身
        return this;
    }
}
```

显然，facts 流是以 Observable 的形式来进行表述的。系统中的其他一些部分接收 API 调用或 Web 请求，然后对发生的事情做出反应（比如向客户收取信用卡费用）并存储 fact（领域事件）。系统的其他组成部分（甚至其他的系统）可以通过订阅流消费这些 fact，然后基于任何角度构建当前系统状态的快照。代码非常简单：每个 ReservationEvent 从投影数据存储中加载一个 Reservation。如果找不到 Reservation，就意味着这是该 UUID 关联的第一个事件，所以就会从一个空的 Reservation 开始，然后将 ReservationEvent 传递给 Reservation 对象。Reservation 可以对自身进行更新以便于应对任意类型的 fact，随后，将 Reservation 保存回去。

需要注意，投影是独立于 fact 的，它们可以使用任意类型的存储机制，甚至只保存在内存之中。另外，你甚至可以让多个投影消费相同的 fact 流，但是构建出不同的快照。例如，可以有一个 Accounting 对象消费相同的 fact 流，但是它只关心现金的流入和流出；另外一个投影可能只关心 FraudDetectedfact，用来汇总欺诈相关的信息。

关于事件溯源的简短介绍有助于你理解 groupBy() 操作符的用处。在一段时间之后，样例对 Reservation 投影更新的速度落后了，它无法与 fact 生成的速度保持同步。数据存储可以很容易地处理并发读取和更新，所以可以尝试并行处理 fact，如下所示。

```
Observable<ReservationEvent> facts = factStore.observe();

facts
        .flatMap(this::updateProjectionAsync)
        .subscribe();

//...

Observable<ReservationEvent> updateProjectionAsync(ReservationEvent event) {
```

```
        //可能是异步的
    }
```

这个样例以并行的方式消费 facts，或者更精确地说：接收是序列化的，但是处理（在 updateProjectionAsync() 中）可能是异步的。updateProjectionAsync() 会更新投影中 Reservation 对象的状态。但是，仔细看一下 updateProjection() 的实现方式，很快就会发现这里可能会有竞态条件：两个线程会消费不同的事件，但是可能会修改同一个 Reservation 并试图进行存储。这样，第一个更新会被覆盖掉，状态就丢失了。技术上，可以尝试乐观锁定，但是另外一个问题依然存在：fact 的顺序无法保证。如果面临的是两个不相关的 Reservation 实例（具有不同的 UUID），那么就不会有这种问题。但是将多个 fact 应用到同一个 Reservation 时，如果应用 fact 的顺序与它们实际发生的顺序不一致，那么后果将是灾难性的。

这就是 groupBy() 能够发挥作用的地方。它会基于某个 key 将流切分为多个并行流，每个流包含具备给定 key 的事件。本例想要将所有关于预定信息的一个 fact 大型流切分为很多小流，每个小流只会发布与特定 UUID 相关的事件，如下所示。

```
Observable<ReservationEvent> facts = factStore.observe();

Observable<GroupedObservable<UUID, ReservationEvent>> grouped =
        facts.groupBy(ReservationEvent::getReservationUuid);

grouped.subscribe(byUuid -> {
    byUuid.subscribe(this::updateProjection);
});
```

这个样例包含了多个新的构造。首先，得到上游的 Observable<ReservationEvent> 并根据 UUID（ReservationEvent::getReservationUuid）对其进行分组。你可能认为 groupBy() 应该会返回一个 List<Observable<ReservationEvent>> 列表，毕竟将一个流转换成了多个。但是这种假设是不能成立的，因为 groupBy() 无法得知上游的流会生成多少个不同的 key（UUID）。因此，结果必须是在运行时生成的：一发现新的 UUID，就发布新的 GroupedObservable<UUID, ReservationEvent> 并推送这个 UUID 相关的事件。所以，显而易见，外层的数据结构必须是 Observable。

但是，这个 GroupedObservable<UUID, ReservationEvent> 又是什么呢？ GroupedObservable 是 Observable 的一个简单子类，它与标准的 Observable 契约不同，它会返回这个流中所有事件所属的一个 key（在样例中也就是 UUID）。发布的 GroupedObservable 的数量范围是从一（这种情况下所有的事件具有相同的 key）到所有事件数量的总和（如果上游中的每个事件都有唯一的 key）。在这种情况下，嵌套的 Observable 不会像以往那样糟糕。订阅外层 Observable 时，发布的每个值实际上是另外一个可以订阅的 Observable（GroupedObservable）。例如，每个内部流可以提供彼此关联的事件（比如具有相同的关联 ID），但是内部流之间是不相关的，可以分别进行处理。

3.4.3 下一步要学习什么

RxJava 内置了很多操作符，其中有一些会在第 6 章进行阐述。但是，讲解所有的 API 并没有太大的意义，并且非常耗时间。同时，这种详尽的描述随着版本更迭也会过时。但是我

们应该对操作符可以做什么，以及是如何运行的有一个基本的了解。接下来让我们学习编写自定义操作符。

3.5 编写自定义的操作符

你现在只是接触了 RxJava 可用操作符中的一些皮毛，但本书通篇学下来，你会掌握更多这方面的知识。除此之外，操作符真正的威力源于它们的组合。遵循 UNIX **小巧且锋利的工具**的理念，[8] 每个操作符每次只进行一个很小的转换。本节首先会介绍 `compose()` 操作符，它能对更小的操作符进行流畅的组合。随后介绍 `lift()` 操作符，借助它能够编写全新的自定义操作符。

3.5.1 借助compose()重用操作符

首先看一个样例。出于某些原因，我们需要转换一个上游的 `Observable`，转换之后只接收偶数条目，而丢弃所有其他条目。6.1 节将会介绍 `buffer()` 操作符，使用这个操作符能够让该任务变得非常简单（`buffer(1, 2)` 几乎完全满足要求）。但是，在以下代码中先假装并不知道这个操作符，通过组合几个操作符也能很容易地实现该功能。

```
import org.apache.commons.lang3.tuple.Pair;

//...

Observable<Boolean> trueFalse = Observable.just(true, false).repeat();
Observable<T> upstream = //...
Observable<T> downstream = upstream
        .zipWith(trueFalse, Pair::of)
        .filter(Pair::getRight)
        .map(Pair::getLeft);
```

首先，样例生成了一个无穷的 `Observable<Boolean>`，它会交替发布 `true` 和 `false`。以上代码创建了只有两个条目、固定的 `[true, false]` 流，然后通过 `repeat()` 操作符无穷重复。`repeat()` 会拦截上游的完成通知，拦截到该通知之后并不会传递给下游，而是重新进行订阅。因此，`repeat()` 操作符并不能保证循环生成相同的事件序列，但是如果上游是一个简单的固定流，那么就能重复生成相同的事件流。参见 7.1.4 节，了解与之类似的 `retry()` 操作符。

我们将上游的 `Observable` 与这个无穷的 `true` 和 `false` 流进行 `zipWith()` 操作。但是，压缩操作需要组合两个条目的函数。在其他语言中，这非常容易实现。在 Java 中，需要借助 Apache Commons Lang 库，它提供了一个简单的 `Pair` 类。此时，我们有了一个用 `Pair<T, Boolean>` 值组成的流，每个值的右侧（`Pair` 由**左侧**和**右侧**组件组成）是 `true` 或 `false`。下一步，使用 `filter()` 过滤所有的 `pair`，只保留右侧为 `true` 的条目，也就是丢弃所有偶数的 `pair`。最后，拆开 `pair` 对象，丢弃 `Boolean` 类型的值，只保留 `T` 类型的值（`getLeft()`）。如果你不想依赖第三方库，那么替换实现如下所示。

注 8：Andrew Hunt 和 David Thoma 编著的《程序员修炼之道——从小工到专家》。

```
import static rx.Observable.empty;
import static rx.Observable.just;

//...

upstream.zipWith(trueFalse, (t, bool) ->
                bool ? just(t) : empty())
        .flatMap(obs -> obs)
```

乍看上去，`flatMap()` 有些奇怪，似乎没有做任何重要的事情。`zipWith()` 转换之后，会返回一个 `Observable`（包含一个元素或者为空），这会形成 `Observable<Observable<T>>`。按照这种方式来使用 `flatMap()`，就能避免嵌套了。毕竟 `flatMap()` 中的 lambda 表达式会为每个输入元素返回一个 `Observable`，这个输入元素恰好也是 `Observable`。

不管你选择使用哪种实现方式，它们都非常难以重用。如果需要重用**每个奇数元素**组成的运算符序列，你要么复制 – 粘贴这些代码，要么创建一个如下所示的工具方法。

```
static <T> Observable<T> odd(Observable<T> upstream) {
    Observable<Boolean> trueFalse = just(true, false).repeat();
    return upstream
            .zipWith(trueFalse, Pair::of)
            .filter(Pair::getRight)
            .map(Pair::getLeft)
}
```

但是，这样就无法流畅地链接操作符了，换句话说，你不能写成 `obs.op1().odd().op2()` 了。与 C#（这也是 Reactive Extensions 起源的地方）和 Scala（通过 implicits）不同，Java 并不允许扩展方法。但是，内置的 `compose()` 操作符实现了非常接近的功能。`compose()` 接收一个函数作为参数，这个函数通过一系列的操作符转换上游的 `Observable`。以下代码是它在实践中使用的样例。

```
private <T> Observable.Transformer<T, T> odd() {
    Observable<Boolean> trueFalse = just(true, false).repeat();
    return upstream -> upstream
            .zipWith(trueFalse, Pair::of)
            .filter(Pair::getRight)
            .map(Pair::getLeft);
}

//...

//[A, B, C, D, E...]
Observable<Character> alphabet =
        Observable
                .range(0, 'Z' - 'A' + 1)
                .map(c -> (char) ('A' + c));

//[A, C, E, G, I...]
alphabet
        .compose(ClassName:odd)
        .forEach(System.out::println);
```

`odd()` 函数会返回一个 `Transformer<T, T>`，也就是从 `Observable<T>` 到 `Observable<T>`（当

然，类型可以不一样）。由于 `Transformer` 本身就是一个函数，因此可以将其替换为 lambda 表达式（`upstream -> upstream...`）。需要注意，`odd()` 函数是在 `Observable` 组装的时候立即执行的，而不是在订阅的时候。有意思的是，如果你想要发布偶数值（第 2 个、第 4 个、第 6 个等）而不是奇数值（第 1 个、第 3 个、第 5 个等），只需要将 `trueFalse` 替换为 `trueFalse.skip(1)` 即可。

3.5.2　使用lift()实现高级操作符

实现自定义操作符是比较麻烦的事情，因为必须要考虑回压（参见 6.2 节）和订阅机制。因此，你最好使用已有的操作符实现需求，而不是自定义操作符。内置操作符经过了更好的测试，功能也经过了验证。但是，如果这些操作符均不满足要求，那么可以借助 `lift()` 元操作符。`compose()` 只能将已有的操作符组合在一起，而借助 `lift()` 几乎可以实现任意的操作符，改变上游事件的流。

`compose()` 转换的是 `Observable`，而 `lift()` 转换的是 `Subscriber`。回忆一下 2.4.1 节介绍过的内容，我们通过 `subscribe()` 订阅一个 `Observable` 的时候，包装回调的 `Subscriber` 实例会被传递给它订阅的 `Observable`，并且导致 `Obsevable` 的 `create()` 方法被调用，在调用时 `subscriber` 会作为参数传递进来（简单概述）。所以，我们每次订阅的时候，一个 `Subscriber` 就会通过所有的操作符传递到原始的 `Observable` 上。显然，在 `Observable` 和 `subscribe()` 之间，可能会有任意数量的操作符，它们会改变流往下游的事件，如下所示。

```
Observable
    .range(1, 1000)
    .filter(x -> x % 3 == 0)
    .distinct()
    .reduce((a, x) -> a + x)
    .map(Integer::toHexString)
    .subscribe(System.out::println);
```

但是，这里有个很有意思的事实：如果你查阅 RxJava 的源码，并将操作符的调用替换为它们的方法体，这个非常复杂的操作符序列看上去就非常有规律了（请注意 `reduce()` 是如何通过 `scan().takeLast(1).single()` 实现的）。

```
Observable
    .range(1, 1000)
    .lift(new OperatorFilter<>(x -> x % 3 == 0))
    .lift(    OperatorDistinct.<Integer>instance())
    .lift(new OperatorScan<>((Integer a, Integer x) -> a + x))
    .lift(    OperatorTakeLastOne.<Integer>instance())
    .lift(    OperatorSingle.<Integer>instance())
    .lift(new OperatorMap<>(Integer::toHexString))
    .subscribe(System.out::println);
```

除了一次操作多个流的操作符（如 `flatMap()`），几乎所有的操作符都是通过 `lift()` 的方式实现的。在最底层调用 `subscribe()` 的时候，RxJava 会创建一个 `Subscriber<String>` 实例并将其传递到直接父节点。这可能是“真正”发布事件的 `Observable<String>`，也可能只是某些操作符的结果，比如样例中的 `map(Integer::toHexString)`。`map()` 本身并不会发布事件，但是它会接收一个 `Subscriber` 实例，而这个实例是想接收事件的。

map() 做的事情（通过 lift() 辅助操作符）就是透明地订阅其父操作符（在上面的样例中，也就是 reduce()）。但是，它不能原样传递它接收到的 Subscriber 实例。这是因为 subscribe() 需要 Subscriber<String>，而 reduce() 预期得到的是 Subscriber<Integer>。这恰好对应 map() 做的事情：将 Integer 转换为 String。因此，map() 操作符会创建一个新的人工 Subscriber<Integer>，每次这个特殊的 Subscriber 接收到事件，它就会调用 Integer::toHexString 函数并通知下游的 Subscriber<String>。

1. 探究map()操作符的内部原理

OperatorMap 类会完成这样的事情：提供一个从下游（child）Subscriber<R> 到上游 Subscriber<T> 的转换。如下代码展现了 RxJava 中的实际实现，其中做了一些易读性方面的简化。

```
public final class OperatorMap<T, R> implements Operator<R, T> {

    private final Func1<T, R> transformer;

    public OperatorMap(Func1<T, R> transformer) {
        this.transformer = transformer;
    }

    @Override
    public Subscriber<T> call(final Subscriber<R> child) {
        return new Subscriber<T>(child) {

            @Override
            public void onCompleted() {
                child.onCompleted();
            }

            @Override
            public void onError(Throwable e) {
                child.onError(e);
            }

            @Override
            public void onNext(T t) {
                try {
                    child.onNext(transformer.call(t));
                } catch (Exception e) {
                    onError(e);
                }
            }
        };
    }
}
```

这里有个值得注意的细节，那就是 T 和 R 泛型的顺序发生了反转。map() 操作符会将上游 T 类型的值转换为 R 类型。但是，操作符的责任是将 Subscriber<R>（来自下游的订阅）转换为 Subscriber<T>（传递给上游的 Observable）。我们期望的是通过 Subscriber<R> 订阅，而 map() 被用来对付 Observable<T>，因此需要 Subscriber<T>。

请确保你已经基本理解了上述 RxJava 源代码中的代码片段。理解 map() 是如何实现的（这是公认的最简单的操作符之一）能够帮助你编写自己的操作符。每次使用 map() 操作一个流时，实际上是使用一个新的 OperatorMap 实例来调用 lift() 并提供 transformer 函数。这个函数会操作上游 T 类型的事件并为下游返回 R 类型的事件。每次用户为你的操作符提供任何自定义的函数 / 转换时，请确保捕获到所有预期之外的异常并将它们通过 onError() 转发到下游。这样也能够确保你会取消对上游流的订阅，避免进一步发布事件。

在有人真正订阅之前，我们在底层几乎不会创建引用 OperatorMap 实例的新 Observable（lift() 与其他的操作符类似，也会创建新的 Observable），而 OperatorMap 实例反过会持有对函数的引用。但是，一旦有人真正订阅，OperatorMap 的 call() 函数就会被调用。这个函数会接收到 Subscriber<String>（例如，包装 System.out::println）并返回另一个 Subscriber<Integer>。后一个 Subscriber 会遍历上游的流来进行处理。

这几乎就是所有操作符的运行原理，无论是内置的，还是自定义的。它们接收一个 Subscriber 并返回另外一个 Subscriber，对事件进行增强并传递下游 Subscriber 想要的任何东西。

2. 第一个操作符

我们想要实现一个操作符，这个操作符能够发布每个奇数元素（第 1 个、第 3 个、第 5 个等）的 toString() 值。如下所示。

```
Observable<String> odd = Observable
        .range(1, 9)
        .lift(toStringOfOdd())
//将会发布字符串"1"，"3"，"5"，"7"和"9"
```

其实，你可以使用内置的操作符完成相同的功能，只是出于教学目的才编写以下自定义操作符的代码。

```
Observable
        .range(1, 9)
        .buffer(1, 2)
        .concatMapIterable(x -> x)
        .map(Object::toString);
```

buffer() 操作符会在 6.1.2 节进行介绍，现在你需要知道的就是 buffer(1, 2) 会将任意的 Observable<T> 转换为 Observable<List<T>>，而每个内部的 List 都只有一个奇数元素，跳过了偶数元素。在拥有了 List(1)、List(3) 等列表组成的流之后，使用 concatMapIterable() 将其重构为一个扁平化的流。但是出于教学的目的，让我们实现一个自定义的操作符，通过一步操作完成这项任务。自定义操作符会处于以下两种状态中的某一种。

- 它从上游流中得到奇数事件（第 1 个、第 3 个、第 5 个等）时，会在调用 toString() 之后将其转发到下游中。
- 它从上游流中得到偶数事件时，丢弃即可。

然后，不断重复这个过程。这个操作符大致如下所示。

```
<T> Observable.Operator<String, T> toStringOfOdd() {
    return new Observable.Operator<String, T>() {

        private boolean odd = true;
```

```
        @Override
        public Subscriber<? super T> call(Subscriber<? super String> child) {
            return new Subscriber<T>(child) {
                @Override
                public void onCompleted() {
                    child.onCompleted();
                }

                @Override
                public void onError(Throwable e) {
                    child.onError(e);
                }

                @Override
                public void onNext(T t) {
                    if(odd) {
                        child.onNext(t.toString());
                    } else {
                        request(1);
                    }
                    odd = !odd;
                }
            };
        }
    };
}
```

对 `request(1)` 的调用将在 6.2.4 节进行介绍。现在你可以这样理解：`Subscriber` 请求事件的一个子集时，比如只取前两个（`take(2)`），RxJava 在内部会通过调用 `request(2)` 只请求符合该数量的数据。这个请求会传递给上游，我们就会只得到 `1` 和 `2`。这里会丢弃 `2`（偶数），但是又必须要为下游提供两个事件，因此，需要请求一个额外的事件（`request(1)`）并添加进来，这样就会得到 `3`。RxJava 实现了一个非常复杂的名为**回压**的机制，这种机制允许订阅者只请求符合其处理能力的事件，避免生产者的输出超过消费者的处理能力。6.2 节将会讨论这个话题。

> 在 RxJava 中，`null` 是一个合法的事件，这其实说不上是好事还是坏事。也就是说，`Observable.just("A", null, "B")` 会像其他流一样正常运行。在设计自定义操作符和使用操作符的时候，你需要考虑这一点。但是，传递 `null` 通常被视为不符合习惯用法，你应该使用包装值类型来进行替代。

另外一个可能出现的很有意思的缺陷，就是无法将子 `Subscriber` 作为参数传递给新的 `Subscriber`，如下所示。

```
<T> Observable.Operator<String, T> toStringOfOdd() {
    //有问题的
    return child -> new Subscriber<T>() {
        //...
    }
}
```

Subscriber 这个无参的构造函数能够正常运行，操作符的运行似乎也没有问题。但是，看一下在遇到无穷流时会发生什么，如下所示。

```
Observable
        .range(1, 4)
        .repeat()
        .lift(toStringOfOdd())
        .take(3)
        .subscribe(
        System.out::println,
        Throwable::printStackTrace,
        () -> System.out.println("Completed")
    );
```

我们构造了一个会发布 ("1"、"2"、"3"、"4"、"1"、"2"、"3"...) 的无穷数字流，应用操作符 ("1"、"3"、"1"、"3"...)，并只取前三个值。这完全没有问题，也不应该会失败，毕竟流是延迟执行的。但是，从 new Subscriber(child) 构造函数中移除 child，Observable 在接收到 1、3、1 后并没有发出完成的通知。这里到底发生了什么呢？

take(3) 操作符只会请求前三个值，并且在此之后想要调用 unsubscribe()。令人遗憾的是，取消订阅的请求并不会被发送到原始的流上，这样原始的流会持续生成值。更糟糕的是，这些值会被自定义操作符处理并传递给下游 Subscriber（take(3)），而下游其实已经不再监听了。先将实现细节放到一边，根据经验，在编写自定义操作符时，要将下游 Subscriber 作为构造函数的参数传递给新的 Subscriber。无参构造函数很少被用到，简单的操作符也不太需要它。

在编写自定义操作符代码时，这些问题仅仅是冰山一角而已。幸而，我们通过内置的机制能够实现绝大多数的场景。

3.6 小结

RxJava 真正的威力在于它的操作符。数据流的声明式转换是非常安全的，并且具有表述性和灵活性。凭借函数式编程强大的用户基础，在评估是否采用 RxJava 时，操作符起决定性的作用。掌握内置操作符是成功使用这个库的关键。本章没有把所有的操作符都介绍到，6.1 节将会介绍更多内容。但是，到目前为止，你应该对 RxJava 能够做什么，以及当它不能直接完成某些任务时该如何进行增强有整体的了解。

第 4 章

将反应式编程应用于已有应用程序

托马什·努尔凯维茨（Tomasz Nurkiewicz）

不管是全新的应用程序还是遗留的代码库，如果要引入新的库、技术或范式，都需要经过仔细的考虑，RxJava 也不例外。本章将回顾普通 Java 应用程序中的一些模式和架构，看看 Rx 应该如何提供帮助。这个过程并不简单，需要转变思维方式，小心地从命令式转换为函数式和反应式风格。目前，Java 项目中的很多库只是让应用程序膨胀，但是并没有带来太大的回报。在本章中，你将了解到 RxJava 如何简化传统项目，以及它为遗留平台带来的收益。

相信你已经对 RxJava 非常兴奋了。内置的操作符和简洁性使得 Rx 成为进行事件流转换的强大工具。但是，如果明天到办公室，你会发现工作环境中没有流，也没有来自证券交易所的实时事件。你在应用程序中很难找到事件，它只是 Web 请求、数据库和外部 API 的混合物。你迫不及待地想要将新的 RxJava 用到某个地方，而不是简单的 Hello world。但是，现实生活中看起来并没有适合使用 Rx 的用例。不过，在架构一致性和健壮性方面，RxJava 被视为重要的进步。你不必彻底采用反应式风格，这样风险太大，并且需要大量的前期工作。但是，Rx 可以在任意层中引入，而不会破坏整个应用程序。

本章将介绍一些通用的应用程序模式和方式。借助它们，你可以使用 RxJava 以非侵入的方式增强应用程序，这里需要重点关注数据库查询、缓存、错误处理和周期性的任务。你在栈的不同位置添加的 RxJava 越多，架构就会越一致。

4.1　从集合到Observable

除非平台是基于近期的 JVM 框架构建的，如 Play、Akka Actors 或 Vert.x，否则你的栈很可能一面是 Servlet 容器，另一面是 JDBC 或 Web Service。在这两者之间，会有不同数量

的层实现业务逻辑。我们不会一次性全部重构这些层，而是从一个简单的样例开始。下面的类代表了一个简单的存储库（repository），它将我们从数据库中抽取了出来。

```
class PersonDao {

    List<Person> listPeople() {
        return query("SELECT * FROM PEOPLE");
    }

    private List<Person> query(String sql) {
        //...
    }

}
```

先将实现细节放到一边，以上代码与 Rx 有什么关系呢？到目前为止，我们讨论了上游系统推送过来的异步事件，还涉及了对它们的订阅。这个普通的 Dao 是如何与此相关联的？Observable 不仅是将事件推送至下游的管道。你可以将 Observable<T> 视为一个数据结构，与 Iterable<T> 相对应。它们都持有 T 类型的条目，但是提供了完全不同的接口。所以，你能够很容易地对它们进行互换，这不足为奇。

```
Observable<Person> listPeople() {
    final List<Person> people = query("SELECT * FROM PEOPLE");
    return Observable.from(people);
}
```

我们对已有的 API 进行了重大变更。根据系统规模的不同，这种不兼容性可能是一个大问题。因此，应该尽快将 RxJava 引入到你的 API。显然，这里使用的是已有的应用程序，因此这种思路并不可行。

4.2 BlockingObservable：脱离反应式的世界

如果你将 RxJava 与已有的、阻塞式的、命令式的代码组合使用，那么可能需要将 Observable **转换**为简单的集合。这种转换令人非常不爽，它需要在 Observable 上阻塞，等待它的完成。在 Observable 完成之前，无法创建集合。BlockingObservable 是一个特殊的类型，借助它能够很容易地在非反应式环境中使用 Observable。在使用 RxJava 的时候，BlockingObservable 应该是最后的选择，但是在组合阻塞代码和非阻塞代码时，它必不可少。

我们在第 3 章重构过 listPeople() 方法，让它返回 Observable<People> 而不是 List。从任何意义上来讲，Observable 都不是 Iterable，所以代码无法编译通过。我们想要循序渐进，而不想进行大规模的重构，所以变更的范围越小越好。客户端代码可能会如下所示。

```
List<Person> people = pesonDao.listPeople();
String json = marshal(people);
```

我们可以将 marshal() 想象为从 people 集合**拉取**（pull）数据，并将它们序列化为 JSON。事实上，情况不再是这样了，我们不能在需要时简单地从 Observable 中**拉取**条目。Observable 负责生成（**推送**）条目，并且如果有订阅者的话，还会通知他们。这种根本性的变化很容易通过 BlockingObservable 来规避。这个非常便利的类完全独立于

Observable，能够通过 Observable.toBlocking() 方法获取。阻塞变种的 Observable 有着表面上类似的方法，比如 single() 或 subscribe()。但是，在没有为异步 Observable 做好准备的阻塞环境中，BlockingObservable 要便利得多。BlockingObservable 上的操作符一般会阻塞（等待），直到底层的 Observable 完成。这与 Observable 中的主要概念严重冲突，即所有操作都是异步、延迟、动态执行的。例如，Observable.forEach() 会异步接收来自 Observable 的事件，但是 BlockingObservable.forEach() 会发生阻塞，直到处理完所有事件并完成流。同时，异常也不再以值（事件）的方式进行传递，而是在调用线程中重新抛出。

在以下案例中，我们想要将 Observable<Person> 转换为 List<Person>，从而限制重构的范围。

```
Observable<Person> peopleStream = personDao.listPeople();
Observable<List<Person>> peopleList = peopleStream.toList();
BlockingObservable<List<Person>> peopleBlocking = peopleList.toBlocking();
List<Person> people = peopleBlocking.single();
```

为了阐述代码在执行过程中都发生了什么，我有意显式保留了所有的中间状态。在重构为 Rx 之后，API 返回的是 Observable<Person> peopleStream。这个流可能是完全反应式、异步和事件驱动的，这与我们的需求完全不匹配：我们需要的是静态的 List。第一步，将 Observable<Person> 转换为 Observable<List<Person>>。这个延迟执行的操作符会在内存中缓冲所有的 Person 事件，直到接收到 onCompleted() 事件。此时，将会发布一个 List<Person> 类型的事件，其中包含了所有可见的事件，如图 4-1 的弹珠图所示。

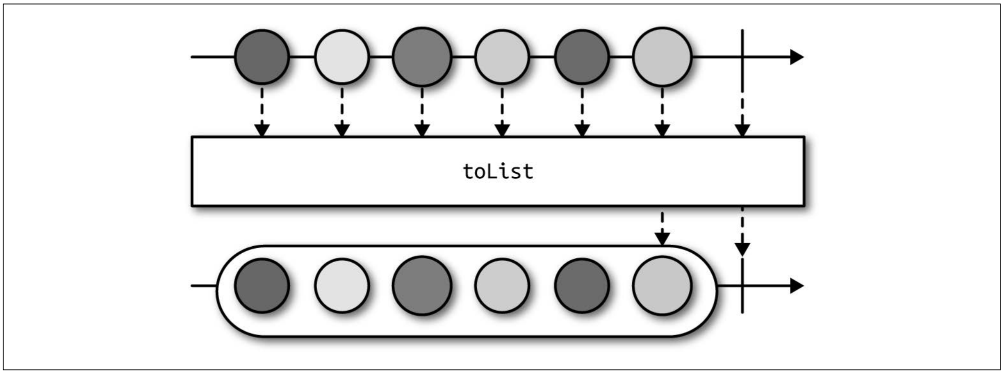


图 4-1

所形成的流在发布完一个 List 条目之后立即完成。同样，这个操作符是异步的，它并不会等待所有的事件，而是以延迟执行的方式缓冲所有的值。看上去有些怪异的 Observable<List<Person>> peopleList 随后被转换成了 BlockingObservable<List<Person>> peopleBlocking。只有必须为异步 Observable 提供阻塞、静态视图的时候，才应该使用 BlockingObservable。Observable.from(List<T>) 会将基于拉取模式的集合转换为 Observable，而 toBlocking() 做的事情恰好相反。你可能会问为什么要为阻塞和非阻塞操作符提供两个抽象。RxJava 的作者指出，明确底层操作符的同步和异步特征非常重要，只依赖 JavaDoc 是靠不住的。使用两个完全不相关的类型能够确保始终使用的都是恰当的数据结构。除此之外，BlockingObservable 应该是你最后的“武器”。正常情况下，你

应该尽可能长地组合和链接普通的 `Observable`。但是，为了完成这个练习，让我们先从 `Observable` 退出来。最后一个操作符 `single()` 会丢弃所有的 `Observable`，并抽取有且仅有的一个条目，这个条目预期从 `BlockingObservable<T>` 接收。类似的操作符 `first()` 会返回 `T` 类型的值并丢弃剩余的内容。而 `single()` 在结束之前要确保底层的 `Observable` 不会有正在等待的事件。这意味着 `single()` 会阻塞等待 `onCompleted()` 回调。如下展示了和前述代码功能相同的片段，只不过这一次将所有的操作符都链接了起来。

```
List<Person> people = personDao
    .listPeople()
    .toList()
    .toBlocking()
    .single();
```

你可能认为以上代码只是封装和拆封 `Observable`，而没有特别明确的目的。但是，这只是第一步。下一个转换将会引入一些延迟执行的功能。现在的代码总是执行 `query("...")` 并使用 `Observable` 对其进行包装。按照定义，`Observable` 是延迟执行的（尤其是 cold 类型的 `Observable`）。如果没有人订阅，它们只代表不会发布任何值的流。大多数情况下，你可以调用返回 `Observable` 的方法，而且只要你不订阅，任何工作都不会做。`Observable` 类似于 `Future`，因为它承诺未来会出现一个值。但是，只要你不发出请求，cold 类型的 `Observable` 甚至不发布值。从这个角度来说，`Observable` 更加类似于 `java.util.function.Supplier<T>`，即按需生成 `T` 类型的值。hot 类型的 `Observable` 与之不同，不管你是否监听，它都会发布值，但是现在我们先不考虑这种类型。仅仅存在 `Observable` 并不意味着会有后台任务或副作用，这与 `Future` 不同，`Future` 几乎总是意味着有某些并发操作在执行。

4.3　拥抱延迟执行

那么，如何让 `Observable` 变成延迟执行的呢？最简单的技术就是使用 `defer()` 对立即执行的 `Observable` 进行包装，如下所示。

```
public Observable<Person> listPeople() {
    return Observable.defer(() ->
        Observable.from(query("SELECT * FROM PEOPLE")));
}
```

`Observable.defer()` 会接收一个 lambda 表达式（工厂），这个表达式会生成 `Observable`。底层的 `Observable` 是立即执行的，所以我们想延迟它的创建。`defer()` 直到最后一刻才会实际创建 `Observable`，即有人订阅它的时候。这意味着会有一些很有意思的事情。因为 `Observable` 是延迟执行的，所以调用 `listPeople()` 不会有什么副作用，也几乎没有性能损耗。此时，还没有进行数据库查询。你可以将 `Observable<Person>` 视为一个承诺，但是还没有进行任何后台处理。注意，此时没有涉及异步行为，仅仅是延迟执行。这类似于在 Haskell 编程语言中，值会延迟到绝对需要的时候才进行计算。

如果你以前从来没有使用过函数式语言进行编程，可能会不理解为何延迟执行如此重要、如此具有颠覆性。事实证明，这种行为非常有用，可以大大提高功能实现的质量和自由度。例如，你不再需要关注要获取哪些资源以及何时以何种顺序获取。RxJava 只会在绝对需要时才加载它们。

比如，我们都见过如下的后备回退（fallback）机制。

```
void bestBookFor(Person person) {
    Book book;
    try {
        book = recommend(person);
    } catch (Exception e) {
        book = bestSeller();
    }
    display(book.getTitle());
}

void display(String title) {
    //...
}
```

你可能觉得这样的构造并没有什么问题。在本例中，我们想要为某个人推荐最好的图书，但是如果失败，则优雅降级，为其展现最畅销的图书。这里的假设前提是获取最畅销图书更快，而且可以进行缓存。但是，如果能够声明式地添加错误处理，try-catch 代码块就不会混淆真正的逻辑了吗？

```
void bestBookFor(Person person) {
    Observable<Book> recommended = recommend(person);
    Observable<Book> bestSeller = bestSeller();
    Observable<Book> book = recommended.onErrorResumeNext(bestSeller);
    Observable<String> title = book.map(Book::getTitle);
    title.subscribe(this::display);
}
```

到目前为止，我们只是探索 RxJava 的用法，所以将所有的中间值和类型都保留了下来。在实际中，bestBookFor() 可能会如下所示。

```
void bestBookFor(Person person) {
    recommend(person)
            .onErrorResumeNext(bestSeller())
            .map(Book::getTitle)
            .subscribe(this::display);
}
```

这段代码简洁易懂。首先为 person 寻找推荐图书。如果出现错误（onErrorResumeNext），就按照畅销图书的逻辑来进行处理。不管哪个获得成功，map 都会返回图书的标题，然后将其展现出来。onErrorResumeNext() 是一个非常强大的操作符，它会拦截上游中的异常，将这些异常吞噬并订阅提供的**备用** Observable。这就是 Rx 实现 try-catch 子句的方式。本书将会在后面对错误处理部分进行详细介绍（参见 7.1.2 节）。目前只需注意我们是如何延迟对 bestSeller() 的调用的，这样就不用担心在真正的图书推荐功能正常运行的情况下，系统还会去获取畅销图书。

4.4 组合Observable

SELECT * FROM PEOPLE 并不是最先进的 SQL 查询。首先，不应该盲目地抓取所有的列，但是抓取所有的行更具破坏性。我们的旧 API 不能对结果进行分页，只能查看表的一个子

集。在传统的企业级应用程序中，它可能会如下所示。

```
List<Person> listPeople(int page) {
    return query(
            "SELECT * FROM PEOPLE ORDER BY id LIMIT ? OFFSET ?",
            PAGE_SIZE,
            page * PAGE_SIZE
    );
}
```

本书并不是一本关于 SQL 的书，所以先将实现的细节放到一边。这个 API 的作者非常无情：我们没有选择记录范围的自由，只能从基于零的页面编号进行操作。但是，在 RxJava 中，借助延迟执行，我们能够模拟从给定页开始读取整个数据库的过程。

```
import static rx.Observable.defer;
import static rx.Observable.from;


Observable<Person> allPeople(int initialPage) {
    return defer(() -> from(listPeople(initialPage)))
            .concatWith(defer(() ->
                    allPeople(initialPage + 1)));
}
```

这个代码片段会延迟加载数据库记录的初始页，比如获取 10 个条目。如果没有人订阅，甚至连第一个查询都不会被调用。如果有一个订阅者只消费初始的几个元素（比如，`allPeople(0).take(3)`），RxJava 将会自动取消对流的订阅，也不会执行进一步的查询。那么，假设请求 11 个条目，而第一次调用 `listPeople()` 仅能返回 10 个条目，此时会怎样？RxJava 断定初始的 `Observable` 已经耗尽，而消费者依然没有得到满足。幸好，它看到了 `concatWith()` 操作符。这就相当于说：当左侧的 `Observable` 完成时，并不会将完成通知传递给下游，而是订阅右侧的 `Observable`，就像什么事情都没有发生过一样，继续执行后续的逻辑，如图 4-2 的弹珠图所示。

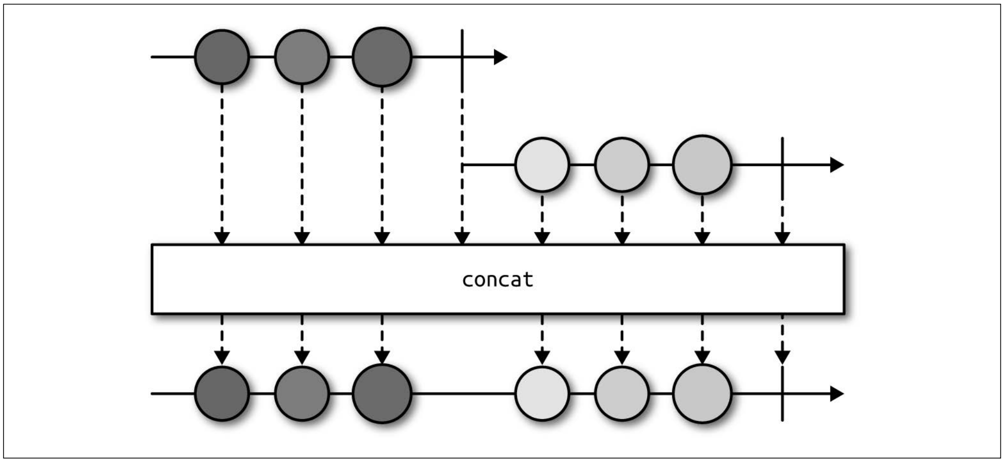


图 4-2

换句话说，concatWith() 能够将两个 Observable 连接在一起，所以当第一个 Observable 完成的时候，第二个 Observable 就会接替它的位置。在 a.concatWith(b).subscribe(...) 中，订阅者首先会接收来自 a 的所有事件，然后再接收来自 b 的所有事件。在本例中，订阅者首先会接收初始的 10 个条目，然后接收后续的 10 个。但是，仔细观察，代码中有一个无穷递归！ allPeople(initialPage) 会调用 allPeople(initialPage + 1)，并且没有任何的停止条件。在大多数语言中，这意味着可能会出现 StackOverflowError，但是这里并不会。同样，对 allPeople() 的调用经常是延迟的，因此停止监听（取消订阅）的时候，递归也就停止了。从技术上讲，concatWith() 依然有可能会导致 StackOverflowError。6.2.4 节会介绍如何处理对传入数据的不同需求。

延迟加载数据块的技术是非常有用的，它能让你专注于业务逻辑，而不必关心底层细节。我们已经看道，即便是在很小的范围内采用 RxJava，也会带来一些收益。按照 Rx 的思路来设计 API 并不会影响整体的架构，因为随时都可以回退到 BlockingObservable 和 Java 集合。但是最好广泛地采用 RxJava，减少对 BlockingObservable 和 Java 集合的使用。

延迟分页和连结

借助 RxJava，会有更多方式来实现延迟分页。如果你思考一下这个问题，加载分页数据的最简单方式就是将所有内容都加载进来，然后按需进行提取。这种做法听起来有点傻，但是借助于延迟执行的特性，其实是可行的。首先，生成所有可能出现的页面编号，然后分别加载每页的数据，如下所示。

```
Observable<List<Person>> allPages = Observable
            .range(0, Integer.MAX_VALUE)
            .map(this::listPeople)
            .takeWhile(list -> !list.isEmpty());
```

如果没有 RxJava，上述的代码会占用大量的时间和内存，基本上相当于把整个数据库加载到了内存中。但是，因为 Observable 是延迟执行的，所以还没有任何对数据库的查询。除此之外，如果发现一个空的数据页，就意味着后续的数据页也会是空的（已经到达了表的底部）。因此，我们使用 takeWhile()，而不是 filter()。为了将 allPages 扁平化为 Observable<Person>，可以使用 concatMap()（参见 3.1.5 节）。

```
Observable<Person> people = allPages.concatMap(Observable::from);
```

concatMap() 需要一个从 List<Person> 到 Observable<Person> 的转换，每个数据页都会执行这个转换。作为替代方案，还可以尝试 concatMapIterable()。它做的是相同的事情，只不过，转换要针对每个上游值返回一个 Iterable<Person>（在这里，上游值恰好已经是 Iterable<Person> 了）。

```
Observable<Person> people = allPages.concatMapIterable(page -> page);
```

不管你采用哪种方式，所有对 Person 对象的转换都是延迟执行的。只要你限制想要处理的记录数量（比如 people.take(15)），Observable<Person> 就会尽可能地往后推迟对 listPeople() 的调用。

4.5 命令式并发

在企业级应用程序中，显式的并发并不常见。大多数情况下，每个请求都会由单个线程处理。同一个线程要完成如下工作。

- 接收 TCP/IP 连接。
- 解析 HTTP 请求。
- 调用控制器或 Servlet。
- 阻塞对数据库的调用。
- 处理结果。
- 编码响应（比如以 JSON 格式）。
- 将原始字节推送至客户端。

如果后端要发起多个独立的请求，比如访问数据库，那么这种分层的模型会影响用户的延迟。它们是序列化执行的，然而可以很容易地并行处理。除此之外，扩展性也会受到影响。例如，在 Tomcat 的执行器（executor）中，默认有 200 个负责处理请求的线程，这意味着处理的并发连接不能超过 200 个。如果流量突然暴增，传入的连接将会排队，服务器就会出现更高的延迟。但是，这种情况不会持续下去，Tomcat 最终会开始拒绝传入的流量。第 5 章将用大量的篇幅（参见 5.1.2 节）介绍如何处理这个相当尴尬的缺点。就目前来说，我们还是继续关注传统的架构。在一个线程中执行请求处理的各个步骤也有一些益处，比如能够提升缓存的本地化以及减少同步的损耗。[1] 令人遗憾的是，在典型的应用程序中，因为整体的延迟是每层延迟的总和，所以一个有故障的组件可能会对整体的延迟产生负面影响。[2] 此外，有时许多步骤是相互独立的，可以并发执行。例如，调用多个外部 API 或执行多个独立的 SQL 查询。

JDK 对并发提供了良好的支持，尤其是 Java 5 提供的 `ExecutorService` 和 Java 8 提供的 `CompletableFuture`。但是，它们并没有得到应有的广泛使用。例如，以下是一个没有任何并发功能的程序。

```
Flight lookupFlight(String flightNo) {
    //...
}

Passenger findPassenger(long id) {
    //...
}

Ticket bookTicket(Flight flight, Passenger passenger) {
    //...
}

SmtpResponse sendEmail(Ticket ticket) {
    //...
}
```

注 1：事实上，RxJava 凭借线程亲和性，试图停留在事件循环模型中的同一线程上，以利用这一点。

注 2：参见 8.2.3 节。

客户端代码如下所示。

```
Flight flight = lookupFlight("LOT 783");
Passenger passenger = findPassenger(42);
Ticket ticket = bookTicket(flight, passenger);
sendEmail(ticket);
```

同样，这是非常典型的阻塞式代码，与众多应用程序中的代码都很类似。但是，如果从延迟的角度来看，上述的代码片段可以分为 4 个步骤。前两个步骤相互独立，只有第三个步骤（bookTicket()）需要 lookupFlight() 和 findPassenger() 的返回值。这里显然有机会利用并发的优势，但是，开发人员很少采用这种方式，因为这需要比较复杂的线程池、Future 以及回调。但是，如果 API 已经兼容 Rx 了，你可以简单地将遗留的阻塞式代码包装到 Observable 中，就像本章开篇那样。

```
Observable<Flight> rxLookupFlight(String flightNo) {
    return Observable.defer(() ->
            Observable.just(lookupFlight(flightNo)));
}

Observable<Passenger> rxFindPassenger(long id) {
    return Observable.defer(() ->
            Observable.just(findPassenger(id)));
}
```

从语义上讲，rx- 方法其实以相同的方式完成了相同的任务，换言之，它们默认都是阻塞的。从客户端的角度看，除了 API 更加冗长之外，我们其实没有得到任何好处。

```
Observable<Flight> flight = rxLookupFlight("LOT 783");
Observable<Passenger> passenger = rxFindPassenger(42);
Observable<Ticket> ticket =
        flight.zipWith(passenger, (f, p) -> bookTicket(f, p));
ticket.subscribe(this::sendEmail);
```

无论是传统的阻塞程序，还是使用 Observable 的程序，它们的运行方式完全相同。上述代码片段默认是延迟执行的，但是操作的顺序依然非常重要。首先，我们创建 Observable<Flight>，默认情况下，此时并不会执行任何操作。在有人明确要求一个 Flight 之前，Observable 只是一个延迟执行的占位符。我们已经介绍过，这是 cold 类型的 Observable 非常有价值的一个特性。Observable<Passenger> 的情况完全相同。现在我们有了 Flight 和 Passenger 类型的两个占位符，但是还没有产生任何的副作用。此时，并没有执行任何的数据库查询或 Web 服务调用。如果从这里决定停止处理，其实并没有执行任何多余的工作。

为了处理 bookTicket()，我们需要具体的 Flight 和 Passenger 实例。最先想到的做法就是使用 toBlocking() 操作符阻塞这两个 Observable。但是，我们应该尽可能地避免阻塞，从而减少资源（尤其是内存）的消耗，支撑更高的并发性。另一个比较糟糕的解决方案就是通过 .subscribe() 方法订阅 flight 和 passenger Observable，然后以某种方式等待它们的回调完成。如果 Observable 是阻塞的，实现起来非常简单；但是如果回调是异步的，你就需要同步一些全局的状态，以等待这两者完成，这种方式很快就会变成一个噩梦。同时，内嵌的 subscribe() 也不符合习惯的用法，通常情况下，你想要对一个信息流（用例）只

进行一次订阅。在 JavaScript 中，回调机制能够比较好地运行的原因在于它只有一个线程。同时订阅多个 `Observable` 的习惯用法是 `zip` 和 `zipWith`。你可能认为 `zip` 是将两个独立数据流中的元素两两连接起来的一种方式。但是更常见的场景是，`zip` 仅仅用于将两个单条目的 `Observable` 连接起来。`ob1.zip(ob2).subscribe(...)` 本质上意味着 `ob1` 和 `ob2` 都完成（二者各自发布完一个事件）的时候，它才会接收到一个事件。所以，当你看到 `zip` 的时候，很可能是有人只针对两个或更多的 `Observable` 执行了一个**连接**（join）步骤，这两个 `Observable` 会有分叉（fork）执行路径。`zip` 是异步等待两个或多个值的一种方式，不管哪个值最后出现。

我们再回到 `flight.zipWith(passenger, this::bookTicket)`（使用了方法引用来替代显式 lambda 表达式，从而更加简短）。这里我保留了所有的类型信息，而不是将表达式流畅地连接在一起，这样做的原因是想让你关注返回的类型。在 `flight` 和 `passenger` 就绪之后，`flight.zipWith(passenger, ...)` 并不会简单地调用回调，它会返回一个新的 `Observable`，你可能会立即意识到这是一个延迟的数据占位符。到目前为止，我们没有启动任何的计算，只是将几个数据结构包装在了一起，没有触发任何的行为。只要没有人订阅 `Observable<Ticket>`，RxJava 就不会运行任何后端代码。而订阅是在最后一个语句中完成的：`ticket.subscribe()` 方法会显式请求 `Ticket`。

应该在何处订阅？

在领域代码中，要关注 `subscribe()` 位于何处。通常，你的业务逻辑只是一直组合 `Observable`，并将它们返回给某个框架或者脚手架层。实际的订阅是由 Web 框架或某些胶水代码在幕后完成的。自行调用 `subscribe()` 也算不上糟糕的实践，但是将订阅推迟得越远越好。

为了理解执行的流程，从下往上观察是一种非常有帮助的方法。我们订阅了 `ticket`，因此 RxJava 必须透明地订阅 `flight` 和 `passenger`。此时，真正的业务逻辑才会执行。因为两个 `Observable` 都是 cold 类型的，并且没有涉及并发，所以对 `flight` 的订阅会在调用线程中触发 `lookupFlight()` 阻塞方法。当 `lookupFlight()` 完成的时候，RxJava 就可以订阅 `passenger` 了。此时，它已经通过同步的 `flight` 接收到 `Flight` 实例。`rxFindPassenger()` 会以阻塞的方式调用 `findPassenger()` 并接收一个 `Passenger`。经过这个连接点之后，数据会往下游流动。`Flight` 和 `Passenger` 实例通过提供的 lambda 表达式（`bookTicket`）被结合起来，传递给 `ticket.subscribe()`。

听上去这里有不少工作要做，运行方式在本质上和开始的阻塞式代码并没有区别。但是，现在我们不需要修改任何逻辑就能声明式地应用并发了。如果业务方法返回 `Future<Flight>`（或者 `CompletableFuture<Flight>`，没有本质区别），其实系统已经为我们做出了两个决策。

- 底层对 `lookupFlight()` 的调用已经开始，这里没有任何延迟执行的空间。我们不会在这个方法上阻塞，但是工作已经启动。
- 我们对并发没有任何控制权。方法的具体实现决定了 `Future` 任务是在线程池调用，还是为每个请求建立一个新的线程。

RxJava 给了用户更多的控制权。Observable<Flight> 在实现的时候没有将并发考虑进去，但我们在后续的操作中依然可以使用并发。现实中的 Observable 一般都已经是异步的了，但是在个别情况下，还是需要为已有的 Observable 添加并发功能。在遇到同步 Observable 时，可以自由决定线程机制的是 API 的消费者，而不是 API 的实现者。上述功能都是通过 subscribeOn() 操作符实现的，如下所示。

```
Observable<Flight> flight =
    rxLookupFlight("LOT 783").subscribeOn(Schedulers.io());
Observable<Passenger> passenger =
    rxFindPassenger(42).subscribeOn(Schedulers.io());
```

我们可以在订阅之前的任何地方插入 subscribeOn() 操作符，并提供一个所谓的 Scheduler 实例。本例中使用了 Schedulers.io() 工厂方法，不过也可以使用自定义的 ExecutorService 并通过 Scheduler 进行简单包装。订阅发生时，传递给 Observable.create() 的 lambda 表达式会在提供的 Scheduler 中执行，而不是在客户端线程中。4.9.1 节将会深度研究调度器，目前可以将 Scheduler 视为一个线程池。

那么 Scheduler 如何改变程序运行时的行为？ zip() 操作符能够订阅两个或更多的 Observable，并等待形成一个配对（或元组）元素。订阅异步发生时，所有上游的 Observable 都可以并发地调用其阻塞式代码。如果你在运行程序时调用 ticket.subscribe()，lookupFlight() 和 findPassenger() 会立即并发执行。这两个 Observable 中较慢的一个发布值之后，bookTicket() 将会立即执行。

既然提到了程序执行的快慢，如果给定的 Observable 在给定时间内未发布任何值，你还可以声明式地设置一个超时，如下所示。

```
rxLookupFlight("LOT 783")
    .subscribeOn(Schedulers.io())
    .timeout(100, TimeUnit.MILLISECONDS)
```

和以前一样，如果出现错误，这些错误会向下游传递，而不是随意抛出。所以，如果 lookupFlight() 方法的耗时超过了 100 毫秒，最终将会形成 TimeoutException，而不会给下游每个订阅者都发布某个值。我们将在 7.1.3 节中详细介绍 timeout() 操作符。

假设 API 已经是 Rx 驱动的，我们不用花费太大的力气就能让两个方法并发执行。但是 bookTicket() 依然有点美中不足，它返回的是 Ticket，毫无疑问是阻塞式的。尽管订票的执行过程可能会非常迅速，但是将其按照 Rx 的方式进行声明也是值得的，这样会让 API 更易于演化。演化可能意味着添加对并发的支持，或者用于完全非阻塞的环境（参见第 5 章）。将非阻塞 API 转换为阻塞 API 非常容易，只需调用 toBlocking()。而反方向的转换通常更具挑战性，需要很多额外的资源。同时，对于像 rxBookTicket() 这样的方法，很难预测其演化方式，如果它们接触网络或文件系统，甚至数据库，那么就值得使用 Observable 进行包装，以便在类型级别上就表明可能会出现延迟。

```
Observable<Ticket> rxBookTicket(Flight flight, Passenger passenger) {
    //...
}
```

但是，现在 zipWith() 返回的是一个看上去很诡异的 Observable<Observable<Ticket>>，

并且无法再编译代码。根据经验，每当你看到双重包装的类型（比如 Optional<Optional<...>>），就意味着在某些地方缺失了对 flatMap() 的调用。这里同样如此。zipWith() 会接收成对（或更通用的元组）出现的事件，以这些事件为参数应用某个函数，然后将结果原样放到下游 Observable 中。这就是为什么我们最初看到的是 Observable<Ticket>，而现在看到的是 Observable<Observable<Ticket>>，这里的 Observable<Ticket> 是我们提供的函数的执行结果。解决这个问题的方式有两种，第一种方式就是使用 zipWith 返回一个中间配对类型，如下所示。

```
import org.apache.commons.lang3.tuple.Pair;

Observable<Ticket> ticket = flight
        .zipWith(passenger, (Flight f, Passenger p) -> Pair.of(f, p))
        .flatMap(pair -> rxBookTicket(pair.getLeft(), pair.getRight()));
```

如果使用第三方的 Pair 类型不足以遮蔽流，那么方法引用也可以完成这项功能：Pair::of。但是在这里，相对于节省几次键盘输入，让类型信息可见是更有价值的。毕竟，阅读代码的次数要比编写代码的次数多一些。替代中间配对类型的一种方案就是使用 flatMap，并为其传递一个恒等式函数。

```
Observable<Ticket> ticket = flight
        .zipWith(passenger, this::rxBookTicket)
        .flatMap(obs -> obs);
```

这个 obs -> obs lambda 表达式看起来什么事情都没有做，至少将其应用到 map() 操作符中的时候是这样。但是，flatMap() 会将一个函数应用到 Observable 中的每个值，所以这个场景中的函数会接收 Observable<Ticket> 作为参数。随后，结果并未直接放到最终形成的流中，这和使用 map() 是一样的。相反，返回值（类型为 Observable<T>）会被“扁平化”。这样形成的就是 Observable<T>，而不是 Observable<Observable<T>>。使用调度器进行处理的时候，flatMap() 操作符会变得更加强大。你也许认为 flatMap() 只是用来解决 Observable<Observable<...>> 嵌套问题的一个语法技巧，其实它还能提供更加基础的功能。

Observable.subscribeOn() 的用例

我们难免会认为 subscribeOn() 是在 RxJava 中实现并发的恰当工具。这个操作符的确能够实现这一点，但尽量还是不要使用它（以及后文描述的 observeOn()）。在现实中，Observable 来源于异步源，所以根本就没有必要进行自定义的调度。本章使用 subscribeOn() 只是为了展现如何升级已有的应用程序，从而有选择性地使用反应式原则。但是，在实践中，Scheduler 和 subscribeOn() 应该是最后的“武器”，所以它们并不那么常见。

4.6 flatMap()作为异步链接操作符

在样例应用程序中，必须要通过电子邮件发送一个 Ticket 列表。在这个过程中，我们必须要注意以下 3 点。

(1) 这个列表可能会很长。

(2) 发送一封邮件可能会耗费几毫秒，甚至几秒的时间。
(3) 在出现故障时，应用程序必须保持平稳地运行，但是最后要报告哪些 ticket 没有成功投递。

最后一项需求迅速排除了简单的 `tickets.forEach(this::sendEmail)` 方式，因为这种方式会立即抛出异常，并且不会继续投递 ticket。实际上，异常机制是类型系统中的一个非常糟糕的后门，就像回调一样，想要以更加健壮的方式对它们进行管理时，它们都不是非常友好。这就是 RxJava 将它们显式地建模为特殊类型通知的原因，本书会在后面介绍这些内容。回到错误处理相关的需求，代码大致如下所示。

```
List<Ticket> failures = new ArrayList<>();
for(Ticket ticket: tickets) {
    try {
        sendEmail(ticket);
    } catch (Exception e) {
        log.warn("Failed to send {}", ticket, e);
        failures.add(ticket);
    }
}
```

但是，前面两个需求或行为指南尚未得到解决。我们没有理由在一个线程中序列化地发送电子邮件。按照传统的方式，我们可以使用 `ExecutorService pool`，将每封电子邮件提交为一个独立的任务。

```
List<Pair<Ticket, Future<SmtpResponse>>> tasks = tickets
    .stream()
    .map(ticket -> Pair.of(ticket, sendEmailAsync(ticket)))
    .collect(toList());

List<Ticket> failures = tasks.stream()
    .flatMap(pair -> {
        try {
            Future<SmtpResponse> future = pair.getRight();
            future.get(1, TimeUnit.SECONDS);
            return Stream.empty();
        } catch (Exception e) {
            Ticket ticket = pair.getLeft();
            log.warn("Failed to send {}", ticket, e);
            return Stream.of(ticket);
        }
    })
    .collect(toList());

//----------------------------------

private Future<SmtpResponse> sendEmailAsync(Ticket ticket) {
    return pool.submit(() -> sendEmail(ticket));
}
```

所有使用 Java 的开发人员对这些代码应该已经非常熟悉了。但是，它看上去非常冗长和复杂。首先，需要遍历 `tickets`，并将它们提交到一个线程池中。准确地说，我们调用 `sendEmailAsync()` 辅助方法，将对 `sendEmail()` 的调用提交到一个线程池中，执行过程会

包装到一个 Callable<SmtpResponse> 中。更准确地说，Callable 的实例被先放到线程池前面的一个无界（默认情况下）队列中。如果任务提交的速度太快，它们将无法及时得到处理。这里缺乏一种减缓提交速度的机制，这也是反应式流和回压致力于解决的问题（参见 6.2 节）。

因为之后我们需要一个 Ticket 实例以防出现故障，所以必须跟踪哪个 Future 负责哪个 Ticket，这里再次以 Pair 来表示。在实际的生产代码中，你应该考虑采用更有意义的专门的容器，比如 TicketAsyncTask 值对象。我们将所有这样的配对信息收集起来，并在下一轮进行处理。此时，线程池中已经并发运行多个 sendEmail() 调用了，这正是我们要达到的目标。第二个循环遍历所有的 Future，并试图通过阻塞（get()）和等待完成的方式来解除对它们的引用。如果 get() 成功返回，将会跳过这个 Ticket。但是，如果出现异常，将会返回与该任务关联的 Ticket 实例，这样就能知道它失败了，稍后再报告它。Stream.flatMap() 允许返回零个或一个元素（实际上可以是任意数量），而 Stream.map() 通常需要一个元素。

你可能想问，为何需要两个循环而不是如下的一个循环呢？

```
//警告：尽管使用了线程池，但代码依然是序列化执行的
List<Ticket> failures = tickets
        .stream()
        .map(ticket -> Pair.of(ticket, sendEmailAsync(ticket)))
        .flatMap(pair -> {
            //...
        })
        .collect(toList());
```

如果你不理解 Java 8 的 Stream 是如何运行的，就很难发现这里有一个非常有意思的 bug。因为流与 Observable 类似，它们都是延迟执行的，所以只有在请求终端操作（terminal operation）的时候（例如 collect(toList())），才会针对底层集合中的每个元素依次执行操作。这意味着启动后台任务的 map() 操作并没有针对所有 ticket 立即执行，而是每次只执行一个元素，交替使用 flatMap()。除此之外，我们实际上启动了一个 Future，阻塞等待，然后启动第二个 Future，阻塞等待，以此类推。这里需要一个中间的集合对象是为了强制执行，而不是为了代码的清晰和可读性。毕竟，List<Pair<Ticket, Future<SmtpResponse>>> 类型已经谈不上什么可读性了。

这里涉及很多工作，并且出现错误的可能性非常高，所以开发人员在日常工作中不愿意使用并发代码也就不足为奇了。如果有一个异步任务的池，并且我们想在任务完成的时候对它们进行处理，那么可以使用 JDK 中不太为人所知的 ExecutorCompletionService。另外，Java 8 引入了 CompletableFuture（参见 5.4 节），它是完全反应式和非阻塞的。那么，RxJava 在这个场景中能够发挥什么作用？首先，假设发送电子邮件的 API 已经被更新为使用 RxJava，如下所示。

```
import static rx.Observable.fromCallable;

Observable<SmtpResponse> rxSendEmail(Ticket ticket) {
    //较为少见的同步Observable
    return fromCallable(() -> sendEmail())
}
```

这里还没有涉及并发，它只是将 sendEmail() 包装进了一个 Observable 中。这种 Observable 并不常见。正常情况下，我们会在实现中使用 subscribeOn()，以便于 Observable 默认采取异步的运行方式。在这里，可以像前面一样遍历所有的 tickets。

```
List<Ticket> failures = Observable.from(tickets)
    .flatMap(ticket ->
        rxSendEmail(ticket)
            .flatMap(response -> Observable.<Ticket>empty())
            .doOnError(e -> log.warn("Failed to send {}", ticket, e))
            .onErrorReturn(err -> ticket))
    .toList()
    .toBlocking()
    .single();
```

> Observable.ignoreElements()
>
> 在以上的样例中，很容易看到内层的 flatMap() 忽略了 response 并返回了一个空的流。在这样的场景中，flatMap() 就有点大材小用了，更有效的方式是 ignoreElements()。ignoreElements() 会忽略上游发布的值，只转发 onCompleted() 或 onError() 通知。因为我们忽略实际的响应，只处理错误，所以这里的 ignoreElements() 能够运行得非常好。

我们感兴趣的内容都在外层 flatMap() 中。如果只是使用 flatMap(this::rxSendEmail)，代码也可以运行，只不过 rxSendEmail 引发的任何故障都会终结整个流。但是，我们想要"捕获"所有发布出来的错误，将其收集起来供后续使用。我们使用了与 Stream.flatMap() 类似的技巧：如果 response 能够成功发布，就将其转换为一个空的 Observable。它的基本含义就是丢弃成功的 ticket。但是，如果遇到故障，样例会返回引发故障的 ticket。额外的 doOnError() 回调允许将异常以日志的形式记录下来。当然，也可以将日志记录添加到 onErrorReturn() 操作符中，但是我们发现这种关注点分离的方式更符合函数式的风格。

为了与之前的实现保持兼容，我们将 Observable 转换为 Observable<List<Ticket>>、BlockingObservable<List<Ticket>>、toBlocking()，最终得到 List<Ticket>（single()）。有意思的是，BlockingObservable 依然是延迟执行的。toBlocking() 操作符本身并不会在订阅底层流的时候就强制执行，它甚至不会阻塞。订阅以及后续的迭代和发送电子邮件，会延迟到调用 single() 的时候才执行。

需要注意，如果将外层的 flatMap() 替换为 concatMap()（参见 3.1.5 节和 3.4.1 节），我们将会遇到与前文提及的 JDK 中的 Stream 类似的 bug。flatMap（或 merge）会立即订阅所有的内部流。与之相反，concatMap（或 concat）则会依次订阅每个内部 Observable。并且只要没有人真正订阅 Observable，它就不会开展任何工作。

到目前为止，一个带有 try-catch 的 for 循环被替换成了更难阅读、更复杂的 Observable。但是，为了将序列化代码转换为多线程计算，我们只需要再加一个操作符，如下所示。

```
Observable
        .from(tickets)
        .flatMap(ticket ->
                rxSendEmail(ticket)
                        .ignoreElements()
```

```
        .doOnError(e -> log.warn("Failed to send {}", ticket, e))
        .onErrorReturn(err -> ticket)
        .subscribeOn(Schedulers.io()))
```

它没有太多的侵入性，你甚至可能很难发现它的存在。额外的 subscribeOn() 操作符会让每个单独的 rxSendMail() 都在一个特定的 Scheduler（本例中是 io()）中执行，这是 RxJava 的优势之一。在线程方面，它没有预设立场，默认同步执行，但是它能够实现无缝甚至透明的多线程功能。当然，这并不意味着你可以在任意位置安全地注入调度器。只不过，它的 API 更加简洁，抽象层级也更高。4.9 节将更详细地探讨调度器。现在，你只需要记住 Observable 默认是同步的。但是，我们可以很轻松地改变这种行为，将并发功能用到往常我们认为不可能出现的地方。这对于现存的遗留应用程序很有价值，借助这种功能，可以轻松地对其进行优化。

如果你从头开始实现 Observable，将它们封装起来，并默认实现异步化更符合习惯用法。这就意味着要将 subscribeOn() 直接放到 rxSendEmail() 中，而不是放到外部。否则，你可能就要使用额外的一层调度器来包装已经异步化的流了。当然，如果 Observable 背后的生产者已经是异步化的，那么流就不用绑定任何的特定线程。除此之外，应该尽可能推迟对 Observable 的订阅，一般这会发生在外部世界的 Web 框架附近。这会大大改变你的思维方式，因为整个业务逻辑都是延迟执行的，直到有人真正想要看到结果的时候才会运行。[3]

4.7 使用Stream代替回调

传统的 API 大多数情况下是阻塞的，这意味着它们会强迫你同步等待结果。这种方式也能运行得非常好，至少在你使用 RxJava 之前。但是，如果数据需要从 API 的生产者推送到消费者，阻塞式的 API 就特别容易出现问题，而这正是 RxJava 能够发挥作用的地方。这样的例子数不胜数，API 的设计师也采用了各种各样的方式。我们通常需要提供某种形式的回调供 API 调用，它们经常被称为事件监听器（event listener）。最常见的场景之一就是 Java 消息服务（Java Message Service，JMS）。消费 JMS 一般需要实现一个类，每次有消息传入，应用程序的服务器或容器就会得到通知。我们可以将这种相对简单的监听器替换为可组合的 Observable，后者会更加健壮和灵活。传统的监听器看上去就像下面这个类，这里使用了 Spring 框架中的 JMS 支持，但是解决方案与具体技术无关。

```
@Component
class JmsConsumer {

    @JmsListener(destination = "orders")
    public void newOrder(Message message) {
        //...
    }
}
```

JMS 消息到达的时候，JmsConsumer 类必须要决定该如何对它进行处理。一般在消息消费者里面会调用一些业务逻辑。新的组件想要接收这样的消息通知时，它必须适当地修改 JmsConsumer。与之不同的是，Observable<Message> 可以被任何人订阅。除此之外，RxJava

注 3：对比 Haskell 的延迟评估表达式。

操作符是完全通用的，可以进行映射、过滤和组合。使用 Subject 是将基于推送、回调的 API 转换为 Observable 的最简单的方式。每次有新的 JMS 消息需要投递，我们就将消息推送给一个 PublishSubject。从外部看，它与一个普通的 hot 类型的 Observable 非常相似。

```
private final PublishSubject<Message> subject = PublishSubject.create();

@JmsListener(destination = "orders", concurrency="1")
public void newOrder(Message msg) {
    subject.onNext(msg);
}

Observable<Message> observe() {
    return subject;
}
```

需要记住的是，Observable<Message> 是 hot 类型的，它们一旦被消费就会开始发布 JMS 消息。如果此时没有人订阅它，消息就会丢失。ReplaySubject 可以作为替代方案解决这个问题，但是它会缓存从应用程序启动开始的所有事件，所以不太适合长期运行的进程。如果真的有订阅者必须接收到所有的消息，那么要确保它在 JMS 消息监听器初始化之前就进行订阅。除此之外，我们的消息监听器还有一个 concurrency="1" 参数，它能确保 Subject 不会在多个线程中被调用。作为替代方案，你还可以使用 Subject.toSerialized()。

补充一下，Subject 易于上手，但是使用一段时间之后很容易出现问题。在这个特定场景中，我们可以很容易地将 Subject 替换为更惯用的 RxJava Observable，后者可以直接调用 create()。

```
public Observable<Message> observe(
    ConnectionFactory connectionFactory,
    Topic topic) {
    return Observable.create(subscriber -> {
        try {
            subscribeThrowing(subscriber, connectionFactory, topic);
        } catch (JMSException e) {
            subscriber.onError(e);
        }
    });
}

private void subscribeThrowing(
        Subscriber<? super Message> subscriber,
        ConnectionFactory connectionFactory,
        Topic orders) throws JMSException {
    Connection connection = connectionFactory.createConnection();
    Session session = connection.createSession(true, AUTO_ACKNOWLEDGE);
    MessageConsumer consumer = session.createConsumer(orders);
    consumer.setMessageListener(subscriber::onNext);
    subscriber.add(onUnsubscribe(connection));
    connection.start();
}

private Subscription onUnsubscribe(Connection connection) {
    return Subscriptions.create(() -> {
        try {
```

```
                connection.close();
            } catch (Exception e) {
                log.error("Can't close", e);
            }
        });
    }
```

JMS API 提供了两种从代理（broker）接收消息的方式：一种是通过阻塞的 `receive()` 方法以同步的方式接收，另一种是通过 `MessageListener` 进行非阻塞接收。非阻塞的 API 有很多优点，比如占用的资源（如线程和栈内存）更少。同时，它与 Rx 编程风格更加一致。在这里，不再创建 `MessageListener` 实例并从它的内部调用订阅者，而是采用方法引用实现更简洁的语法。

```
consumer.setMessageListener(subscriber::onNext)
```

同时，还要清理资源和处理异常。这个很小的转换层能够让我们很便利地消费 JMS 消息，而不必担心 API 的内部细节。如下是使用流行的 ActiveMQ 消息代理的样例，该代理在本地运行。

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQTopic;

ConnectionFactory connectionFactory =
    new ActiveMQConnectionFactory("tcp://localhost:61616");
Observable<String> txtMessages =
        observe(connectionFactory, new ActiveMQTopic("orders"))
        .cast(TextMessage.class)
        .flatMap(m -> {
            try {
                return Observable.just(m.getText());
            } catch (JMSException e) {
                return Observable.error(e);
            }
        });
```

JMS 和 JDBC 类似，同样因为使用大量的检查型异常 `JMSException` 而广受诟病，即便是针对 `TextMessage` 调用 `getText()` 也不例外。为了恰当地处理错误（参见 7.1 节），我们使用 `flatMap()` 并包装异常。从这里开始，就可以将 JMS 消息流视为其他任何异步和非阻塞的流。值得一提的是，这里使用了 `cast()` 操作符。这个操作符会将上游的事件转换为指定的类型，如果出现失败，将会调用 `onError()`。`cast()` 基本上就是一个特殊的 `map()` 操作符，它的行为类似于 `map(x -> (TextMessage)x)`。

4.8 定期轮询以获取变更

你可能遇到的最糟糕的阻塞式 API 要求轮询变更。这种 API 不会提供任何推送变更的机制，甚至没有回调或永久阻塞机制。这种 API 提供的唯一机制就是请求当前状态，然后由你来判断它是否与之前的状态有差异。RxJava 有一些非常强大的操作符，借助它们，可以将给定的 API 转换为 Rx 风格。接下来考虑一个简单的方法，它会返回一个代表当前状态的值，例如 `long getOrderBookLength()`。为了跟踪它的变化，必须要频繁地调用这个方法

并捕获变更。在 RxJava 中，我们可以借助非常基本的操作符组合实现这一点，如下所示。

```
Observable
        .interval(10, TimeUnit.MILLISECONDS)
        .map(x -> getOrderBookLength())
        .distinctUntilChanged()
```

首先，每 10 毫秒，我们就会人为地生成一个 long 值，使用它作为基础的计时计数器。对于每个这样的值（每 10 毫秒生成一个），都调用 getOrderBookLength()。但是，前面提及的方法并不会变化得那么频繁，而且我们不希望订阅者接收到大量无关的状态变更。幸好，可以只使用 distinctUntilChanged()，RxJava 会透明地跳过 getOrderBookLength() 返回的 long 值，那些值自从上次调用以来没有发生变化，如图 4-3 的弹珠图所示。

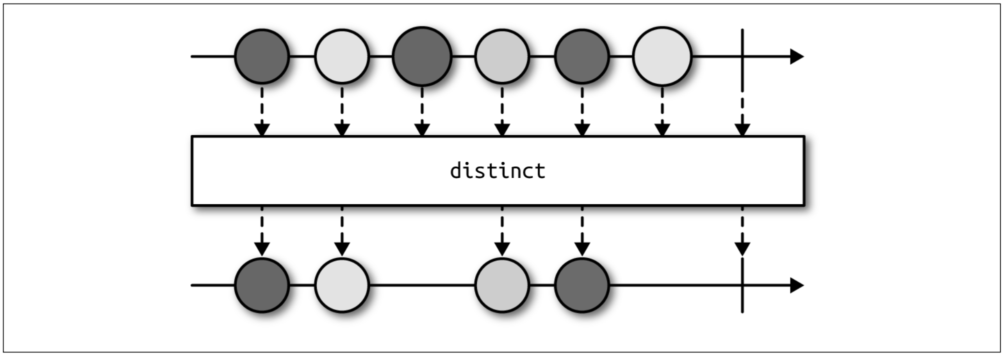


图 4-3

在这个模式的使用上，我们可以更进一步。假设你正在监视文件系统或数据库表的变更。你唯一可以使用的机制就是获取当前文件或数据库记录的快照。每次有新的条目，你构建的 API 要通知所有的客户端。显然，你可以使用 java.nio.file.WatchService 或数据库触发器，但是这里只是作为辅助讲解的样例。这次我们还是阶段性地获取当前状态的一个快照，如下所示。

```
Observable<Item> observeNewItems() {
    return Observable
            .interval(1, TimeUnit.SECONDS)
            .flatMapIterable(x -> query())
            .distinct();
}

List<Item> query() {
    //获取文件系统目录或数据库表的快照
}
```

distinct() 操作符会保留流经它的所有条目的一个记录（参见 3.4.4 节）。如果相同的条目第二次出现，它就会被忽略。这就是我们可以每秒推送相同的 Item 列表的原因。第一次出现的时候，它们会推送给下游所有的订阅者。但是，完全相同的列表在 1 秒后再次出现时，由于所有的条目都出现过了，所以它们会被丢弃。如果在某个时间点，query() 返回的列表包含了一个额外的 Item，那么 distinct() 就会将其传递至下游。但是这个 Item 下

次出现的时候，则会被丢弃。在这种简单的模式下，我们就可以使用定期轮询替换大量的 `Thread.sleep()` 调用和手动缓存。它适用于很多场景，比如文件传输协议（File Transfer Protocol，FTP）轮询、Web 抓取等。

4.9 RxJava的多线程

很多第三方 API 都是阻塞式的，而且我们几乎对它们无能为力。我们可能没有源代码，重写会导致非常高的风险。在这种情况下，必须要学会如何处理阻塞代码，而不是与之缠斗。

RxJava 的一个显著特征就是声明式并发，而不是命令式并发。手动创建和管理线程已经是过去的事情了（参见 A.3 节），大多数人已经开始使用托管线程池了（比如借助 `ExecutorService`）。但是，RxJava 更进一步：`Observable` 可以像 Java 8 中的 `CompletableFuture`（参见 5.4 节）一样是非阻塞的，但是与后者不同，`Observable` 还是延迟执行的。除非进行订阅，否则设计良好的 `Observable` 不会执行任何操作。而 `Observable` 的威力并不仅限于此。

异步的 `Observable` 能够从不同的线程调用 `Subscriber` 回调方法（比如 `onNext()`）。回忆一下 2.4.1 节，我们讨论过如果 `subscribe()` 是阻塞的，它会一直等待，直到所有的通知到达。实际中，大多数的 `Observable` 来自本身就有异步特征的事件源。整个第 5 章都会阐述这样的 `Observable`。即便是 4.7 节中的简单 JMS 样例，也使用了 JMS 规范中的内置的、非阻塞的 API（`MessageListener` 接口）。尽管在类型系统上没有强制要求，但是很多 `Observable` 一开始就是异步的，而且也应当假设它们都是异步的。当 `Observable.create()` 中的 lambda 表达式没有任何异步进程或流作为支撑时，阻塞式的 `subscribe()` 方法很少用到。但是，默认情况下（使用 `create()`），所有的逻辑都会在客户端线程（即进行订阅的线程）中执行。如果你只是在 `create()` 回调中直接使用 `onNext()`，那么不会涉及任何的多线程和并发。

如果遇到这种不太符合常规的 `Observable`，我们可以**声明式地**选择所谓的 `Scheduler`，它会被用来发布值。在 `CompletableFuture` 中，我们无法控制底层的线程，API 会做出决策；在最糟糕的情况下，我们甚至不能覆盖它。RxJava 很少独自做这样的决策，它会选择一种安全的默认做法：使用客户端线程，不去涉及多线程。为了更好地介绍本章内容，我们会使用一个非常简单的日志“库”。[4] 它会打印当前线程的信息，以及使用 `System.currentTimeMillis()` 获取的从程序启动开始持续的毫秒数，如下所示。

```
void log(Object label) {
    System.out.println(
        System.currentTimeMillis() - start + "\t| " +
        Thread.currentThread().getName()   + "\t| " +
        label);
}
```

4.9.1 调度器是什么

RxJava 是并发无关的，它本身并没有引入并发。但是，它将一些用来处理线程的抽象暴露给了终端用户。同时，离开了并发，一些特定的操作符（参见 4.9.6 节）也无法正常

注 4：显然，对于任何一个实际的程序，你会使用产品级的日志系统，如 Logback 或 Log4J 2。

运行。你唯一需要关注的是 Scheduler 类，而它非常简单。在理念上，它与 java.util.concurrent 中的 ScheduledExecutorService 非常相似——它能够执行任意的代码块，这些代码可能是在未来运行的。但是，为了满足 Rx 的契约，它提供了一些细粒度的抽象。更深入的信息，你可以参考本节的“调度器实现细节概览”部分。

Scheduler 会在创建特定类型的 Observables 时，与 subscribeOn() 和 observeOn() 操作符协同使用。调度器会创建 Worker 实例，并由后者负责进行调度和运行代码。RxJava 需要调度代码的时候，它会首先请求 Scheduler 提供一个 Worker，然后借助该 Worker 调度后续的任务。后文会有该 API 的样例，但首先来熟悉一下可用的内置调度器。

- Schedulers.newThread()

  每当通过 subscribeOn() 或 observeOn() 请求这个调度器时，它都会启动一个新的线程。newThread() 通常并不是很好的可选方案。这不仅是因为启动线程涉及延迟，还因为这个线程并不能被重用。在这种情况下，必须要为线程预先分配栈空间（通常大约在 1 MB 左右，可以由 JVM 的 -Xss 参数来进行控制），操作系统必须要启动一个新的本机线程。Worker 完成的时候，线程就会终止。只有任务是非常粗粒度的情况下，这种调度器才会有用武之地：这种任务需要耗费很长时间才能完成。但是这样的任务数量很少，所以线程被重用的可能性也很小，参见 A.2 节。在实践中，下述的 Schedulers.io() 通常是更好的选择。

- Schedulers.io()

  这个调度器类似于 newThread()，但是它会回收已启动的线程，这些线程可以用来处理未来的请求。这个实现的运行方式类似于 java.util.concurrent 中具有无限线程池的 ThreadPoolExecutor。每次请求一个新的 Worker，要么启动一个新的线程（并且随后会在一段时间内维持空闲状态），要么就重用空闲的线程。将其命名为 io() 并非巧合。考虑将这个调度器用到 I/O 密集同时需要很少 CPU 资源的任务上。然而，这样的任务可能会耗费相当长的时间等待网络和磁盘。因此，比较好的做法是使用相对比较大的线程池。不过，对于任何类型的无限制资源都要非常小心。如果遇到慢速或无响应的外部依赖，比如 Web 服务，io() 将会启动数量庞大的线程，从而导致应用程序无法响应。参见 8.2 节，了解更多处理这种问题的细节。

- Schedulers.computation()

  如果任务是 CPU 密集型，那么你应该使用计算调度器，这种类型的任务需要计算能力，但是没有阻塞式代码（从磁盘读入、网络访问、休眠、等待锁等）。因为在调度器上执行这样的任务都会希望充分利用 CPU 核心，所以即便并行执行的任务数量大于可用核心的数量，也不会带来太大的价值。鉴于此，computation() 调度器默认会将并行运行的线程数量限制为 availableProcessors() 方法返回的值，这个方法来源于 Runtime.getRuntime() 工具类。

  如果基于某些原因，你需要的线程数量与默认值不相同，那么可以使用 rx.scheduler.max-computation-threads 系统属性。使用数量更少的线程能够确保即便是在高负载的情况下，也会始终有一个或多个 CPU 核心处于空闲状态。这样能够避免 computation() 线程池使服务器处于饱和状态。我们无法让计算线程的数量超过 CPU 核心的数量。

computation() 会在每个线程前面使用一个无界队列，所以如果任务已经调度了，但是当前所有的核心均被占用，那么它们将会排队。在负载高峰时段，这个调度器能够限制线程的数量，但是每个线程前面的队列将会持续增长。

幸好，借助内置的操作符，尤其是 4.9.5 节将要介绍到的 observeOn()，能够确保 Scheduler 不会超载。

- Schedulers.from(Executor executor)

  Scheduler 内部要比 java.util.concurrent 包中的 Executor 更为复杂，所以需要一个独立的抽象。但是它们在理念上非常相似，所以有一个能将 Executor 转换为 Scheduler 的包装器也就在情理之中了，这个包装器使用的是 from() 工厂方法，如下所示。

```
import com.google.common.util.concurrent.ThreadFactoryBuilder;
import rx.Scheduler;
import rx.schedulers.Schedulers;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;

//...

ThreadFactory threadFactory = new ThreadFactoryBuilder()
    .setNameFormat("MyPool-%d")
    .build();
Executor executor = new ThreadPoolExecutor(
    10, //corePoolSize
    10, //maximumPoolSize
    0L, TimeUnit.MILLISECONDS, //keepAliveTime, unit
    new LinkedBlockingQueue<>(1000), //workQueue
    threadFactory
);
Scheduler scheduler = Schedulers.from(executor);
```

这里故意使用了比较冗长的语法来创建 ExecutorService，而没有使用如下更为简洁的版本。

```
import java.util.concurrent.Executors;

//...

ExecutorService executor = Executors.newFixedThreadPool(10);
```

这种方式尽管看上去很有吸引力，但是 Executors 工厂类硬编码了一些默认值。在企业级应用程序中，这种做法是不可行的，甚至可以说是危险的。例如，它使用了无界的 LinkedBlockingQueue，这种队列可以无限增长，如果有大量未完成的任务，这将会导致 OutOfMemoryError。另外，默认的 ThreadFactory 会使用毫无意义的线程名，比如 pool-5-thread-3。在诊断和分析线程转储的时候，恰当的线程名是非常宝贵的工具。从头实现 ThreadFactory 有些麻烦，所以使用了 Guava 的 ThreadFactoryBuilder。如果你对如何优化和更恰当地使用线程池感兴趣，可以参考 A.3 节。对于处理高负载的项目，建议采用精心配置的 Executor 来创建调度器。但是，RxJava 无法控制 Executor 中独立创建的线程，所

以它无法对线程进行锁定（也就是，尽量让相同的任务在同一个线程上执行，从而提升缓存本地化的效果）。这个 Scheduler 只能确保每个 Scheduler.Worker（参见本节的“调度器实现细节概览”部分）会按照顺序来处理事件。

- Schedulers.immediate()

  Schedulers.immediate() 是一个特殊的调度器，它会以阻塞的方式在客户端线程中调用某个任务，而不是采用异步的方式。使用这个调度器并没有太大的意义，除非你的 API 需要提供一个调度器，否则你完全可以直接使用 Observable 的默认行为，根本不涉及任何线程。实际上，通过 immediate() Scheduler 订阅一个 Observable（稍后会详细介绍）的效果通常与不使用任何调度器进行订阅的效果完全一样。一般情况下，要避免使用这个调度器，因为它会阻塞调用，而且用途有限。

- Schedulers.trampoline()

  trampoline() 与 immediate() 非常相似，它也在相同的线程中调度任务，实际上是阻塞式的。但是与 immediate() 不同，使用 trampoline() 时，后续的任务会在所有已调度的任务全部完成之后才开始执行。immediate() 会立即执行给定的任务，而 trampoline() 则会等待当前的任务完成。Trampoline 是函数式编程中的一种模式，允许实现递归，而不会出现无限增长的调用栈。这最好通过一个例子来进行阐述，如下所示。首先使用 immediate()。注意这里没有直接与 Scheduler 进行交互，而是创建了一个 Worker。这对你学习本节的“调度器实现细节概览”部分会有一定帮助。

```
Scheduler scheduler = Schedulers.immediate();
Scheduler.Worker worker = scheduler.createWorker();

log("Main start");
worker.schedule(() -> {
    log(" Outer start");
    sleepOneSecond();
    worker.schedule(() -> {
        log(" Inner start");
        sleepOneSecond();
        log(" Inner end");
    });
    log(" Outer end");
});
log("Main end");
worker.unsubscribe();
```

输出完全符合预期，实际上可以将 schedule() 替换成简单的方法调用。

```
1044    | main  | Main start
1094    | main  |  Outer start
2097    | main  |   Inner start
3097    | main  |   Inner end
3100    | main  |  Outer end
3100    | main  | Main end
```

在 Outer 代码块中，我们使用 schedule() 来调度 Inner 代码块，Inner 代码块会立即被调用并中断对 Outer 任务的调用。Inner 完成之后，控制权会重新回到 Outer。再次强调，这只是以

比较复杂的方式利用 immediate() Scheduler 间接实现了阻塞式的任务调用。但是，如果将 Schedulers.immediate() 替换为 Schedulers.trampoline() 会怎么样？输出将会有很大的差异。

```
1030    | main  | Main start
1096    | main  |  Outer start
2101    | main  |  Outer end
2101    | main  |   Inner start
3101    | main  |   Inner end
3101    | main  | Main end
```

在这里，Outer 完成之后，Inner 才会启动。这是因为 Inner 任务会在 trampoline() Scheduler 排队，而此时该调度器已经被 Outer 任务占据了。Outer 完成之后，队列中的第一个任务（Inner）就会开始执行。我们可以更进一步，确保你理解了它们的差异，如下所示。

```
log("Main start");
worker.schedule(() -> {
    log(" Outer start");
    sleepOneSecond();
    worker.schedule(() -> {
        log(" Middle start");
        sleepOneSecond();
        worker.schedule(() -> {
            log(" Inner start");
            sleepOneSecond();
            log(" Inner end");
        });
        log(" Middle end");
    });
    log(" Outer end");
});
log("Main end");
```

来自 immediate() Scheduler 的 Worker 输出如下所示。

```
1029    | main  | Main start
1091    | main  |  Outer start
2093    | main  |   Middle start
3095    | main  |    Inner start
4096    | main  |    Inner end
4099    | main  |   Middle end
4099    | main  |  Outer end
4099    | main  | Main end
```

来自 trampoline() Worker 的输出如下所示。

```
1041    | main  | Main start
1095    | main  |  Outer start
2099    | main  |  Outer end
2099    | main  |   Middle start
3101    | main  |   Middle end
3101    | main  |    Inner start
4102    | main  |    Inner end
4102    | main  | Main end
```

- `Schedulers.test()`

  这个 Scheduler 只用来进行测试，它不能在生产代码中使用。它的主要优势在于能够任意推进时钟，模拟时间的推移。7.2.2 节会对 TestScheduler 进行更详细的描述。Scheduler 本身并没有太大的意思，如果你想要探寻它们内部是如何运行的，以及如何实现自己的调度器，那么请阅读下面的内容。

调度器实现细节概览

> 本节的内容完全是可选的，如果你对实现细节不感兴趣，可以直接跳到 4.9.2 节。

Scheduler 不仅将任务与它们的执行解耦（一般通过将它们在另外的线程中运行来实现），它还将时钟抽象了出来，这一点在 7.2.1 节将会做进一步阐述。相对于 ScheduledExecutorService 来说，Scheduler 的 API 更简洁一些，如下所示。

```
abstract class Scheduler {
    abstract Worker createWorker();

    long now();

    abstract static class Worker implements Subscription {

        abstract Subscription schedule(Action0 action);

        abstract Subscription schedule(Action0 action,
                                       long delayTime, TimeUnit unit);

        long now();
    }
}
```

RxJava 想要调度一个任务时（一般会在后台执行，但并非必须如此），它必须首先请求一个 Worker 实例。借助 Worker，既可以不加延迟地调度任务，也可以在未来的某个时间点调度任务。Scheduler 和 Worker 都有一个可重写的时间源（now() 方法），这个方法用来决定给定的任务要何时运行。简单地说，你可以将 Scheduler 视为线程池，而将 Worker 视为池中的线程。

Scheduler 和 Worker 的分离是非常必要的。这样就能很容易地实现 Rx 契约要求的一些指导原则，即顺序调用 Subscriber 的方法，而不是并发调用。Worker 的契约提供了这样的功能：在相同 Worker 上调度的任务永远不会并发运行。但是，来自同一个 Scheduler 的独立 Worker 可以很好地并发运行。

不再讨论 API，让我们分析一下已有 Scheduler 的源代码，即 RxAndroid 项目中的 HandlerScheduler。这个 Scheduler 会在 Android 的 UI 线程中运行所有调度任务，更新用户界面的操作也只能在这个线程中运行（参见 8.1 节）。这类似于 Swing 中的事件分发线程（Event Dispatch Thread，EDT），对窗口和组件的大多数更新都要在专门的线程（EDT）中运行。毫不意外的是，针对 Swing 也有一个名为 RxSwing 的项目。

下面的代码片段是从 RxAndroid 抽取出来的一个不太完整的类，只是为了便于知识的讲解。

```
package rx.android.schedulers;

import android.os.Handler;
import android.os.Looper;
import rx.Scheduler;
import rx.Subscription;
import rx.functions.Action0;
import rx.internal.schedulers.ScheduledAction;
import rx.subscriptions.Subscriptions;

import java.util.concurrent.TimeUnit;

public final class SimplifiedHandlerScheduler extends Scheduler {

    @Override
    public Worker createWorker() {
        return new HandlerWorker();
    }

    static class HandlerWorker extends Worker {

        private final Handler handler = new Handler(Looper.getMainLooper());

        @Override
        public void unsubscribe() {
            //马上会实现……
        }

        @Override
        public boolean isUnsubscribed() {
            //马上会实现……
            return false;
        }

        @Override
        public Subscription schedule(final Action0 action) {
            return schedule(action, 0, TimeUnit.MILLISECONDS);
        }

        @Override
        public Subscription schedule(
        Action0 action, long delayTime, TimeUnit unit) {
            ScheduledAction scheduledAction = new ScheduledAction(action);
            handler.postDelayed(scheduledAction, unit.toMillis(delayTime));

            scheduledAction.add(Subscriptions.create(() ->
                    handler.removeCallbacks(scheduledAction)));

            return scheduledAction;
        }
    }
}
```

现在，Android API 的细节并不重要。这里，每次在 HandlerWorker 上调度任务，代码块都会被传递到一个特殊的 postDelayed() 方法中，这个方法会在专门的 Android 线程中执行代码。这样的线程只有一个，因此不仅仅是某个 Worker 内部的事件是序列化执行的，所有 Worker 的事件都是如此。

action 在付诸执行之前，我们使用 ScheduledAction 对它进行包装，它同时实现了 Runnable 和 Subscription。如果可能，RxJava 始终都会延迟执行，这一原则同样适用于调度任务。如果基于某种原因，你决定取消执行给定的 action（这适用于 action 调度到未来的某个时间点执行，而不是立即执行的场景），那么只需要根据 schedule() 方法返回的 Subscription 实例，运行其 unsubscribe() 方法即可。Worker 负责恰当地处理取消订阅的相关工作（至少它会尽最大努力处理好）。

客户端代码也可以从 Worker 级别通过 unsubscribe() 全部取消订阅。这样会取消所有排队的任务并释放 Worker，便于底层的线程稍后进行重用。如下的代码片段增强了 SimplifiedHandlerScheduler，为其添加了 Worker 取消订阅的流程（只包含修改过的代码）。

```
private CompositeSubscription compositeSubscription =
    new CompositeSubscription();

@Override
public void unsubscribe() {
    compositeSubscription.unsubscribe();
}

@Override
public boolean isUnsubscribed() {
    return compositeSubscription.isUnsubscribed();
}

@Override
public Subscription schedule(Action0 action, long delayTime, TimeUnit unit) {
    if (compositeSubscription.isUnsubscribed()) {
        return Subscriptions.unsubscribed();
    }

    final ScheduledAction scheduledAction = new ScheduledAction(action);
    scheduledAction.addParent(compositeSubscription);
    compositeSubscription.add(scheduledAction);

    handler.postDelayed(scheduledAction, unit.toMillis(delayTime));

    scheduledAction.add(Subscriptions.create(() ->
            handler.removeCallbacks(scheduledAction)));

    return scheduledAction;
}
```

我们在 2.3 节中探讨了 Subscription 接口，但是并没有真正展开实现细节。CompositeSubscription 是众多可用实现中的一个，它本身只是子 Subscription 的容器（即**组合**设计模式），在 CompositeSubscription 上取消订阅的操作意味着要取消对所有子 Subscription 的订阅。你还可以添加和移除 CompositeSubscription 管理的子 Subscription。

在我们自定义的 Scheduler 中，CompositeSubscription 用来跟踪前面 schedule() 方法调用形成的所有 Subscription（参见 compositeSubscription.add(scheduledAction)）。另一方面，子 ScheduledAction 还需要知道其父 Subscription（参见 addParent()），这样，操作完成或取消的时候，它就能够将自己移除。否则，Worker 会不断地累积旧的子 Subscription。客户端代码决定不需要某个 Handler Worker 实例时，它会对其取消订阅。取消订阅的操作会传递到所有未完成的子 Subscription（如果存在）。

以上就是对 RxJava 中 Scheduler 的简短介绍。在日常工作中，这些内部细节用处并不大。实际上，将它们设计成这种方式就是让 RxJava 的使用更加直观和可预测。接下来，快速看一下 Scheduler 是如何解决 Rx 中众多的并发问题的。

4.9.2 使用subscribeOn()进行声明式订阅

2.4.1 节介绍过，subscribe() 默认使用客户端的线程。扼要重述，如下是最简单的订阅功能，不涉及任何的线程。

```
Observable<String> simple() {
    return Observable.create(subscriber -> {
        log("Subscribed");
        subscriber.onNext("A");
        subscriber.onNext("B");
        subscriber.onCompleted();
    });
}

//...

log("Starting");
final Observable<String> obs = simple();
log("Created");
final Observable<String> obs2 = obs
        .map(x -> x)
        .filter(x -> true);
log("Transformed");
obs2.subscribe(
        x -> log("Got " + x),
        Throwable::printStackTrace,
        () -> log("Completed")
);
log("Exiting");
```

请注意日志记录语句放在了什么地方，然后仔细看以下输出，尤其是哪个线程调用了打印语句。

```
33  | main  | Starting
120 | main  | Created
128 | main  | Transformed
133 | main  | Subscribed
133 | main  | Got A
133 | main  | Got B
133 | main  | Completed
134 | main  | Exiting
```

注意：语句的顺序完全是可预测的。首先，上述代码片段中的每行代码都在 main 线程中运行，不涉及任何线程池和异步事件的发布。其次，乍看上去，代码的执行顺序可能并不十分清晰。

程序启动的时候，它会首先会打印出 Starting，这一点非常容易理解。在创建 Observable<String> 实例之后，我们看到了 Created 信息。请注意，真正开始订阅后，才会随之出现 Subscribed。如果不调用 subscribe()，Observable.create() 中的代码块就永远不会执行。另外，map() 和 filter() 操作符甚至没有任何可见的副作用，而 Transformed 消息出现在了 Subscribed 之前。

随后，我们接收到所有发布出来的事件以及完成通知。最后，程序打印出 Exiting，然后就可以返回了。这是一个非常有意思的观察结果：我们会认为 subscribe() 只是注册了一个回调，而事件会以异步的方式出现。这可能是默认的假定。但是，在这种场景中，并没有涉及线程，subscribe() 实际上是阻塞的。为什么要以这种方式实现？

在 subscribe() 和 create() 之间有一个内在却隐含的连接。每次在 Observable 上调用 subscribe() 的时候，就会调用它的 OnSubscribe 回调方法（该方法包装了传递给 create() 的 lambda 表达式）。它接收你的 Subscriber 作为参数。默认情况下，这会在同一个线程中执行，并且是阻塞的，换言之，无论在 create() 中做什么，都会阻塞 subscribe()。如果你的 create() 休眠几秒，那么 subscribe() 就会阻塞。如果在 Observable.create() 和你的 Subscriber（作为回调的 lambda 表达式）之间存在操作符，所有的操作符也都会在调用 subscribe() 的线程中执行。RxJava 默认在 Observable 和 Subscriber 之间并没有插入任何的并发基础设施。这背后的原因在于 Observable 一般是由其他并发机制支撑的，比如事件循环或自定义线程，所以 Rx 将控制权完全交给你，而不做任何强加的约定。

上面的观察结果为介绍 subscribeOn() 操作符做好了铺垫。可以将 subscribeOn() 放到原始 Observable 和 subscribe() 之间的任意地方，通过这种方式，声明式地定义的 OnSubscribe 回调方法会在所选的 Scheduler 中执行。不管在 create() 中进行何种操作，这些任务都会由独立的 scheduler 承担，而 subscribe() 不会再阻塞。

```
log("Starting");
final Observable<String> obs = simple();
log("Created");
obs
        .subscribeOn(schedulerA)
        .subscribe(
                x -> log("Got " + x),
                Throwable::printStackTrace,
                () -> log("Completed")
        );
log("Exiting");

35  | main | Starting
112 | main | Created
123 | main | Exiting
123 | Sched-A-0 | Subscribed
124 | Sched-A-0 | Got A
124 | Sched-A-0 | Got B
124 | Sched-A-0 | Completed
```

不知你是否注意到，main 线程在 Observable 发布值之前就已经退出了。从技术上讲，日志信息的顺序无法预测，因为这两个线程是并发执行的：main 会进行订阅并退出，而 Sched-A-0 会在有人订阅后立刻发布事件。schedulerA 和 Sched-A-0 都来源于如下的示例调度器，构建这个调度器就是为了便于进行说明。

```
import static java.util.concurrent.Executors.newFixedThreadPool;


ExecutorService poolA = newFixedThreadPool(10, threadFactory("Sched-A-%d"));
Scheduler schedulerA = Schedulers.from(poolA);

ExecutorService poolB = newFixedThreadPool(10, threadFactory("Sched-B-%d"));
Scheduler schedulerB = Schedulers.from(poolB);

ExecutorService poolC = newFixedThreadPool(10, threadFactory("Sched-C-%d"));
Scheduler schedulerC = Schedulers.from(poolC);

private ThreadFactory threadFactory(String pattern) {
    return new ThreadFactoryBuilder()
        .setNameFormat(pattern)
        .build();
}
```

这些调度器将会用到所有的样例中，不过要记住它们也非常容易。这里有三个独立的调度器，每个都管理着来自 ExecutorService 的 10 个线程。为了让输出更加直观，每个线程池都有不同的命名模式。

在开始之前，你必须要知道，在采用 Rx 的成熟应用程序中，很少会用到 subscribeOn()。正常情况下，Observable 源于本来就已经是异步的源（如 RxNetty，参见 5.1.2 节）或者它们本身就已经使用了调度（如 Hystrix，参见 8.2 节）。subscribeOn() 仅用于一些特殊的场景，也就是你已经知道底层 Observable 是同步的（create() 是阻塞的）时候。但是，相对于在 create() 中手动编写线程相关的代码，subscribeOn() 是一个好得多的方案。

```
//不要这样做
Observable<String> obs = Observable.create(subscriber -> {
    log("Subscribed");
    Runnable code = () -> {
        subscriber.onNext("A");
        subscriber.onNext("B");
        subscriber.onCompleted();
    };
    new Thread(code, "Async").start();
});
```

上述代码混淆了两个概念：生成事件和选择并发策略。Observable 只应该负责事件的生成逻辑，而只有客户端代码才能对并发性做出正确的决策。需要记住，Observable 是延迟执行的，并且是不可变的。在某种意义上讲，subscribeOn() 只会影响下游订阅者。如果有人订阅了同一个 Observable，但是没有使用 subscribeOn()，默认不涉及任何并发。

本章关注的是已有的应用程序以及如何逐步引入 RxJava。在这种情况下，subscribeOn() 操作符非常有用，但是，在你掌握了 Reactive Extensions 并开始大规模使用它们之后，

subscribeOn() 的价值就降低了。在完全反应式的软件栈中，比如 Netflix，几乎不会用到 subscribeOn()，因为所有的 Observable 都已经是异步的了。大多数情况下，Observable 来自异步的事件源，并且默认它们都是异步的。因此，subscribeOn() 的使用场景非常有限，主要用于完善现有 API 和库的工作。我们在第 5 章中将会编写完全异步的应用程序，到那时就不会显式使用 subscribeOn() 和 Scheduler 了。

4.9.3 subscribeOn()的并发性和行为

关于 subscribeOn() 如何运行，有一些很有意思的细节。首先，好奇的读者可能想知道，如果在 Observable 和 subscribe() 之间调用两次 subscribeOn()，将会如何运行？答案非常简单：最靠近原始 Observable 的 subscribeOn() **胜出**。这具有重要的实际意义。如果你正在设计一个 API，并在内部使用了 subscribeOn()，那么客户端代码将无法覆盖你选择的 Scheduler。这可以称得上是精心思考的设计决策，毕竟，只有 API 的设计者最了解哪个 Scheduler 更合适。另一方面，比较好的办法是提供上述 API 的重载版本，从而允许覆盖 API 设计者选择的 Scheduler。

接下来，让我们学习一下 subscribeOn() 的行为。

```
log("Starting");
Observable<String> obs = simple();
log("Created");
obs
        .subscribeOn(schedulerA)
        //很多其他的操作符
        .subscribeOn(schedulerB)
        .subscribe(
                x -> log("Got " + x),
                Throwable::printStackTrace,
                () -> log("Completed")
        );
log("Exiting");
```

如下输出只会显示 schedulerA 的线程。

```
17 | main  | Starting
73 | main  | Created
83 | main  | Exiting
84 | Sched-A-0 | Subscribed
84 | Sched-A-0 | Got A
84 | Sched-A-0 | Got B
84 | Sched-A-0 | Completed
```

有意思的是，对 schedulerA 来说，schedulerB 并非被完全忽略。schedulerB 也会在很短的时间内被用到，不过只是用来在 schedulerA 上调度新的操作，而所有的工作都是由 schedulerA 完成的。因此，多个 subscribeOn() 不仅会被忽略，还会引入一点额外的开销。

说到操作符，本书提到过，在新的 Subscriber 执行时，create() 会在提供的调度器（如果有的话）中执行。但是，该由哪个线程来执行 create() 和 subscribe() 之间的所有转换？默认情况下，所有的操作符都会在相同的线程（调度器）中执行，默认不会涉及并发。

```
log("Starting");
final Observable<String> obs = simple();
log("Created");
obs
        .doOnNext(this::log)
        .map(x -> x + '1')
        .doOnNext(this::log)
        .map(x -> x + '2')
        .subscribeOn(schedulerA)
        .doOnNext(this::log)
        .subscribe(
                x -> log("Got " + x),
                Throwable::printStackTrace,
                () -> log("Completed")
        );
log("Exiting");
```

我们在操作符管道中添加了 doOnNext()，以便查看此时哪个线程处于控制状态。subscribeOn() 的位置无关紧要，它可以紧跟在 Observable 后面，也可以放在 subscribe() 之前。如下的输出结果并不令人意外。

```
20  | main  | Starting
104 | main  | Created
123 | main  | Exiting
124 | Sched-A-0 | Subscribed
124 | Sched-A-0 | A
124 | Sched-A-0 | A1
124 | Sched-A-0 | A12
124 | Sched-A-0 | Got A12
124 | Sched-A-0 | B
124 | Sched-A-0 | B1
124 | Sched-A-0 | B12
125 | Sched-A-0 | Got B12
```

请注意观察如何调用和生成 A、B 事件。这些事件按顺序依次通过调度器的线程，最后到达 Subscriber。很多刚接触 RxJava 的读者会认为，如果将大量线程与 Scheduler 组合使用，RxJava 会自动对事件进行分叉并发处理，并在最后以某种形式将结果联结（join）起来。但事实并非如此。RxJava 会为整个管道只创建一个 Worker 实例（参见 4.9.1 节），主要是为了确保事件能够按顺序进行处理。

这意味着，如果你有一个特别慢的操作符，比如 map() 操作符为了转换流经的事件需要从磁盘读取数据，那么这个成本高昂的操作会在相同的线程中运行。一个糟糕的操作符会减缓整个管道的运行速度，从生产者直到消费者均不能幸免。在 RxJava 中，这是一种反模式，操作符应该是非阻塞的、快速运行的，并且工作内容越单纯越好。

在这方面，flatMap() 可以再次施以援手。与 map() 中的阻塞行为不同，我们可以调用 flatMap() 并异步收集所有的结果。因此，如果想要实现真正的并行，flatMap() 和 merge() 操作符能够达到这一目的。但是，即便是使用 flatMap()，也并非那么简单。假设有一个杂货店（即 RxGroceries），它提供了购买商品的 API，如下所示。

```
class RxGroceries {

    Observable<BigDecimal> purchase(String productName, int quantity) {
        return Observable.fromCallable(() ->
            doPurchase(productName, quantity));
    }

    BigDecimal doPurchase(String productName, int quantity) {
        log("Purchasing " + quantity + " " + productName);
        //实际的逻辑在这里
        log("Done " + quantity + " " + productName);
        return priceForProduct;
    }

}
```

显然，在这里 doPurchase() 的实现无关紧要，只需假设它需要一定的时间和资源才能完成。我们通过添加 1 秒的休眠来模拟业务逻辑，如果 quantity 值更大，休眠时间会更长一点。在实际应用程序中，像 purchase() 方法返回的这种阻塞 Observable 并不常见，但是为了讲解的需要，我们让它保持这种运行方式。购买多件商品的时候，我们想要尽可能地并行处理，并在最后计算所有商品的总价。第一次尝试是徒劳的。

```
Observable<BigDecimal> totalPrice = Observable
        .just("bread", "butter", "milk", "tomato", "cheese")
        .subscribeOn(schedulerA)  //有问题的!!!
        .map(prod -> rxGroceries.doPurchase(prod, 1))
        .reduce(BigDecimal::add)
        .single();
```

最终结果是正确的，它是只包含一个值的 Observable，也就是通过 reduce() 计算得到的总价。对于每种商品，我们调用了 doPurchase()，并将 quantity 设置为一。但是，尽管 schedulerA 由 10 个线程的池作为支撑，代码依然是完全序列化执行的。

```
144  | Sched-A-0 | Purchasing 1 bread
1144 | Sched-A-0 | Done 1 bread
1146 | Sched-A-0 | Purchasing 1 butter
2146 | Sched-A-0 | Done 1 butter
2146 | Sched-A-0 | Purchasing 1 milk
3147 | Sched-A-0 | Done 1 milk
3147 | Sched-A-0 | Purchasing 1 tomato
4147 | Sched-A-0 | Done 1 tomato
4147 | Sched-A-0 | Purchasing 1 cheese
5148 | Sched-A-0 | Done 1 cheese
```

请注意，每种商品都阻塞了后续商品的处理。面包的购买完成之后，会立即处理黄油的购买请求，但是无法在此之前开始进行处理。诡异的是，即便将 map() 替换为 flatMap() 也没有任何的助益，输出完全相同。

```
Observable
        .just("bread", "butter", "milk", "tomato", "cheese")
        .subscribeOn(schedulerA)
        .flatMap(prod -> rxGroceries.purchase(prod, 1))
        .reduce(BigDecimal::add)
        .single();
```

代码之所以没有并行执行，原因就在于这里只有一个事件流，按照设计它们必须序列化运行。否则，你的 Subscriber 就需要感知并发的通知（onNext()、onComplete() 等）了，所以这是一个折中的方案。幸而，惯用的方案与之非常接近。发布商品的主 Observable 不能并行化，但是对于每种商品，当它们从 purchase() 返回时候，我们可以创建新的、独立的 Observable。因为是独立的，也就意味着可以安全地对它们进行并发处理。

```
Observable<BigDecimal> totalPrice = Observable
        .just("bread", "butter", "milk", "tomato", "cheese")
        .flatMap(prod ->
                rxGroceries
                        .purchase(prod, 1)
                        .subscribeOn(schedulerA))
        .reduce(BigDecimal::add)
        .single();
```

你注意到 subscribeOn() 在哪里了吗？主 Observable 并没有做任何事情，所以不需要特殊的线程池。但是，样例为 flatMap() 中创建的每个子流都提供了 schedulerA。每次 subscribeOn() 被调用，Scheduler 都能返回一个新的 Worker，因此也会有一个单独的线程。

```
113  | Sched-A-1 | Purchasing 1 butter
114  | Sched-A-0 | Purchasing 1 bread
125  | Sched-A-2 | Purchasing 1 milk
125  | Sched-A-3 | Purchasing 1 tomato
126  | Sched-A-4 | Purchasing 1 cheese
1126 | Sched-A-2 | Done 1 milk
1126 | Sched-A-0 | Done 1 bread
1126 | Sched-A-1 | Done 1 butter
1128 | Sched-A-3 | Done 1 tomato
1128 | Sched-A-4 | Done 1 cheese
```

最后，样例实现了真正的并发。每个购买的操作都会同时开始，并且最终完成。flatMap() 经过了精心的设计和实现，所以它会收集所有独立流的事件并将它们按顺序推送至下游。但是，正如 3.1.4 节介绍的，此时不能再依赖下游事件的顺序了，它们开始和完成的顺序已经与最初发布的顺序不同了（原始的序列是从面包开始的）。它们顺序抵达 reduce() 操作符的时候，就已经是序列化的了，并且能够按顺序执行。

到现在为止，你应该已经逐渐摆脱了传统的 Thread 模型，并且理解了 Scheduler 是如何运行的。但是，如果你觉得还有困难，可以看看如下的简单类比。

- 没有 Scheduler 的 Observable 就像一个单线程的程序,以阻塞式方法在彼此之间传递数据。
- 具有一个 subscribeOn() 的 Observable 就像在后台 Thread 中启动了一个大型的任务。Thread 中的程序依然是序列化的，但至少它会在后台运行。
- 如果 Observable 使用了 flatMap()，并且每个内部 Observable 都使用了 subscribeOn()，那么其运行方式就像是 java.util.concurrent 中的 ForkJoinPool，每个子流都是分叉执行的，而 flatMap() 是一个安全的**联结**点。

当然，上述类比仅适用于阻塞式的 Observable，而在实际应用程序中，这种 Observable 是比较少见的。如果你的底层 Observable 已经是异步的，那么实现并发仅仅需要理解它们该如何进行组合，以及何时进行订阅。例如，对两个流进行 merge() 操作会并发订阅这两个流，而 concat() 会一直等到第一个流完成才订阅第二个流。

4.9.4 使用groupBy()进行批量请求

不知道你是否注意到了，即便商品的数量始终为一，RxGroceries.purchase() 方法也会接收 productName 和 quantity 这两个参数。如果某些商品在货物清单上出现了多次，意味着对它们有更高的需求，又该怎么办？最简单的实现方式就是多次发送相同的请求，比如，多次发送对鸡蛋的购买请求，每次只买一个。幸好，可以使用 groupBy()，以声明式的方式批量处理这样的请求，而且声明式并发依然适用。

```
import org.apache.commons.lang3.tuple.Pair;

Observable<BigDecimal> totalPrice = Observable
    .just("bread", "butter", "egg", "milk", "tomato",
      "cheese", "tomato", "egg", "egg")
    .groupBy(prod -> prod)
    .flatMap(grouped -> grouped
        .count()
        .map(quantity -> {
            String productName = grouped.getKey();
            return Pair.of(productName, quantity);
        }))
    .flatMap(order -> store
        .purchase(order.getKey(), order.getValue())
        .subscribeOn(schedulerA))
    .reduce(BigDecimal::add)
    .single();
```

这段代码非常复杂，所以在介绍输出内容之前，让我们先快速浏览一下代码的内容。首先，根据商品的名称对其进行分组，所以使用了恒等函数 prod -> prod。它返回的是看上去有点烦琐的 Observable<GroupedObservable<String, String>>，这本身是没有什么问题的。接下来，flatMap() 操作符会接收每个 GroupedObservable<String, String>，后者代表了具有相同名称的所有商品。例如，对于值为 "egg" 的 key，将会对应 ["egg", "egg", "egg"] Observable。如果 groupBy() 使用不同的 key 生成函数，比如 prod.length()，那么相同的数据序列对应的 key 就是 3 了。

此时，在 flatMap() 中，需要构建一个 Pair<String, Integer> 类型的 Observable，它代表了每种唯一的商品及其数量。count() 和 map() 返回的都是 Observable，所以一切都满足需求。第二个 flatMap() 会接收 Pair<String, Integer> 类型的 order，然后进行购买，这一次购买数量可以更大一些。输出看上去已经满足了要求，请注意大订单会更慢一些，但是依然比多次重复请求要快。

```
164  | Sched-A-0 | Purchasing 1 bread
165  | Sched-A-1 | Purchasing 1 butter
166  | Sched-A-2 | Purchasing 3 egg
166  | Sched-A-3 | Purchasing 1 milk
166  | Sched-A-4 | Purchasing 2 tomato
166  | Sched-A-5 | Purchasing 1 cheese
1151 | Sched-A-0 | Done 1 bread
1178 | Sched-A-1 | Done 1 butter
1180 | Sched-A-5 | Done 1 cheese
1183 | Sched-A-3 | Done 1 milk
```

```
1253 | Sched-A-4 | Done 2 tomato
1354 | Sched-A-2 | Done 3 egg
```

如果你认为自己的系统能够从这种或其他的批处理方式中获益，那么可以参考 8.2.4 节。

4.9.5 使用observeOn()声明并发

不管你是否相信，在 RxJava 中有两种方式来声明并发，即前文所述的 subscribeOn() 以及接下来要介绍的 observeOn()。二者看上去非常相似，初学者很容易混淆，但是它们的语义其实非常清晰合理。

subscribeOn() 让我们能够选择使用哪个 Scheduler 来触发 OnSubscribe（create() 中的 lambda 表达式）。因此，create() 中的代码都会被放到一个不同的线程中，例如，可以通过这种方式来避免阻塞主线程。与之不同的是，observeOn() 在被调用之后，能够控制该由哪个 Scheduler 触发下游的 Subscriber。例如，调用 create() 发生在 io() Scheduler 中（通过 subscribeOn(io()) 实现），从而避免阻塞用户界面。但是，更新用户界面的组件必须要在 UI 线程中运行（Swing 和 Android 都有这样的限制），所以我们要在更新 UI 的操作符和订阅者调用之前调用 observeOn()，并将（例如）AndroidSchedulers.mainThread() 传递进来。通过这种方式，我们可以使用某个 Scheduler 来处理 create() 方法以及第一个 observeOn() 之前的所有操作符，使用其他的 Scheduler 来进行一些转换。让我们通过一个例子来进行阐述，如下所示。

```
log("Starting");
final Observable<String> obs = simple();
log("Created");
obs
        .doOnNext(x -> log("Found 1: " + x))
        .observeOn(schedulerA)
        .doOnNext(x -> log("Found 2: " + x))
        .subscribe(
                x -> log("Got 1: " + x),
                Throwable::printStackTrace,
                () -> log("Completed")
        );
log("Exiting");
```

observeOn() 会在管道链的某个地方出现，但是与 subscribeOn() 不同，observeOn() 的位置非常重要。不管在 observeOn() 之前是哪个 Scheduler 在运行操作符（如果存在 Scheduler），但是在该方法之后，就会由提供的 Scheduler 来运行所有的工作。在本例中，没有 subscribeOn()，所以使用默认的 Scheduler（即没有并发），输出如下。

```
23  | main  | Starting
136 | main  | Created
163 | main  | Subscribed
163 | main  | Found 1: A
163 | main  | Found 1: B
163 | main  | Exiting
163 | Sched-A-0 | Found 2: A
164 | Sched-A-0 | Got 1: A
164 | Sched-A-0 | Found 2: B
```

```
164 | Sched-A-0 | Got 1: B
164 | Sched-A-0 | Completed
```

observeOn 之前的所有操作符都是在客户端线程中运行的，这正是 RxJava 的默认行为。但是在 observeOn() 之后，所有的操作符都会在提供的 Scheduler 中运行。如果在管道中同时使用 subscribeOn() 和多个 observeOn()，那么结果看上去就会更明显了，如下所示。

```
log("Starting");
final Observable<String> obs = simple();
log("Created");
obs
        .doOnNext(x -> log("Found 1: " + x))
        .observeOn(schedulerB)
        .doOnNext(x -> log("Found 2: " + x))
        .observeOn(schedulerC)
        .doOnNext(x -> log("Found 3: " + x))
        .subscribeOn(schedulerA)
        .subscribe(
                x -> log("Got 1: " + x),
                Throwable::printStackTrace,
                () -> log("Completed")
        );
log("Exiting");
```

你能预测一下输出吗？注意，observeOn() 之后的所有代码都会在提供的 Scheduler 中运行，当然，这里的所有代码指的是另一个 observeOn() 出现之前的内容。另外，subscribeOn() 可以在 Observable 和 subscribe() 之间的任意位置出现，但是此时它只能影响第一个 observeOn() 出现的位置。输出如下所示。

```
21  | main  | Starting
98  | main  | Created
108 | main  | Exiting
129 | Sched-A-0 | Subscribed
129 | Sched-A-0 | Found 1: A
129 | Sched-A-0 | Found 1: B
130 | Sched-B-0 | Found 2: A
130 | Sched-B-0 | Found 2: B
130 | Sched-C-0 | Found 3: A
130 | Sched-C-0 | Got: A
130 | Sched-C-0 | Found 3: B
130 | Sched-C-0 | Got: B
130 | Sched-C-0 | Completed
```

订阅是在 schedulerA 中发生的，因为这是在 subscribeOn() 中指定的。同时，Found 1 操作符也是在该 Scheduler 中执行的，因此它出现在第一个 observeOn() 之前。之后的事情就比较有意思了。observeOn() 将当前的 Scheduler 切换成了 schedulerB，Found 2 就会使用 schedulerB 了。最后一个 observeOn(schedulerC) 会影响 Found 3 操作符和 Subscriber。需要记住，Subscriber 会在最后一个 Scheduler 的上下文中执行。

如果你想物理解耦生产者（Observable.create()）和消费者（Subscriber），subscribeOn() 和 observeOn() 能够非常好地协作运行。默认情况下，并不存在这样的结构，RxJava 会使用相同的线程。仅有 subscribeOn() 并不够，借助它只能选择不同的线程。observeOn() 更

好一些，但是如果遇到同步的 Observable，将会导致客户端线程阻塞。大多数操作符都是非阻塞的，里面的 lambda 也会非常简短，执行代价并不高，所以在操作符管道中，一般只使用一个 subscribeOn() 和 observeOn()。subscribeOn() 可以放到接近原始 Observable 的位置，以便于提升可读性，而 observeOn() 应该放到接近 subscribe() 的位置。这样，只有 Subscriber 会使用这个特殊的 Scheduler，其他的操作符则会依赖来自 subscribeOn() 的 Scheduler。

如下是一个更高级的程序，它用到了这两个操作符。

```
log("Starting");
Observable<String> obs = Observable.create(subscriber -> {
    log("Subscribed");
    subscriber.onNext("A");
    subscriber.onNext("B");
    subscriber.onNext("C");
    subscriber.onNext("D");
    subscriber.onCompleted();
});
log("Created");
obs
    .subscribeOn(schedulerA)
    .flatMap(record -> store(record).subscribeOn(schedulerB))
    .observeOn(schedulerC)
    .subscribe(
            x -> log("Got: " + x),
            Throwable::printStackTrace,
            () -> log("Completed")
    );
log("Exiting");
```

其中，store() 是一个简单的嵌套操作符。

```
Observable<UUID> store(String s) {
    return Observable.create(subscriber -> {
        log("Storing " + s);
        //这里会有一些繁重的工作
        subscriber.onNext(UUID.randomUUID());
        subscriber.onCompleted();
    });
}
```

事件在 schedulerA 中生成，但是每个事件都是使用 schedulerB 独立处理的，这样能够提升并发性，这在 4.9.3 节介绍过。而最终的订阅却是发生在 schedulerC 中的。相信你现在已经能够理解哪个 Scheduler/ 线程分别执行哪些操作了，但是以防万一，请观察以下输出（这里出于清晰的考虑添加了空行）。

```
26  | main   | Starting
93  | main   | Created
121 | main   | Exiting

122 | Sched-A-0 | Subscribed
124 | Sched-B-0 | Storing A
124 | Sched-B-1 | Storing B
```

```
124  | Sched-B-2 | Storing C
124  | Sched-B-3 | Storing D

1136 | Sched-C-1 | Got: 44b8b999-e687-485f-b17a-a11f6a4bb9ce
1136 | Sched-C-1 | Got: 532ed720-eb35-4764-844e-690327ac4fe8
1136 | Sched-C-1 | Got: 13ddf253-c720-48fa-b248-4737579a2c2a
1136 | Sched-C-1 | Got: 0eced01d-3fa7-45ec-96fb-572ff1e33587
1137 | Sched-C-1 | Completed
```

对于具备 UI 的应用程序来说，observeOn() 尤为重要，在这样的应用程序中，我们不想阻塞 UI 的事件－分派线程。在 Android（参见 8.1 节）或 Swing 中，更新 UI 等操作必须要在特定的线程中执行。但是，如果在这个线程中做太多的事情，将会导致 UI 无法响应。在这种情况下，我们将 observeOn() 放在靠近 subscribe() 的位置，以便在特定 Scheduler（比如 UI 线程）的上下文中调用订阅中的代码。而其他的转换，即便执行成本很低，也应该在 UI 线程之外调用。在服务器端很少用 observeOn()，因为大多数的 Observable 已经内置了并发。这就得出了一个很有意思的结论：RxJava 只使用两个操作符（subscribeOn() 和 observeOn()）就控制了并发，但是你使用 Reactive Extensions 越频繁，在生产代码中看到这两个操作符的机会就越少。

4.9.6　调度器的其他用途

很多操作符默认用到了一些 Scheduler。一般情况下，如果不指定（JavaDoc 会详细说明），就会使用 Schedulers.computation()。例如，delay() 操作符接收上游的事件，并在给定的时间之后将这些事件传递到下游。显然，它不能在等待的时候一直持有原始的线程，所以它必须要使用一个不同的 Scheduler。

```
Observable
        .just('A', 'B')
        .delay(1, SECONDS, schedulerA)
        .subscribe(this::log);
```

如果不提供自定义 schedulerA，delay() 后面所有的操作符都将使用 computation() Scheduler。这本身并没有什么问题。但是，如果你的 Subscriber 在 I/O 上被阻塞了，那么它将会占用一个来自全局 computation() 调度器的 Worker，这有可能会对整个系统产生影响。其他支持自定义 Scheduler 的重要操作符有 interval()、range()、timer()、repeat()、skip()、take()、timeout()，还有一些是本书尚未介绍的。如果你不为这些操作符提供调度器，将会使用 computation() Scheduler，这在大多数情况下是一种安全的默认做法。

掌握调度器对于编写可扩展和安全的 RxJava 代码至关重要。理解 subscribeOn() 和 observeOn() 的差异也是尤为重要的，因为在高负载的场景中，每个任务都必须要非常精准地按照预期来执行。在真正的反应式应用程序中，所有长期运行的操作都是异步的，因此只需要很少的线程和 Scheduler。但是，依然会有 API 或依赖需要阻塞式的代码。

最后，但并非不重要的一点，我们必须要确保下游使用的 Scheduler 能够承受住上游 Scheduler 产生的负载。这种危险的场景会在第 6 章进行阐述。

4.10 小结

本章描述了在传统应用程序中可以替换为 RxJava 框架的一些模式。希望你现在能够理解，高频率的交易或者社交媒体上的帖子更新并不是 RxJava 的唯一用例。实际上，几乎所有的 API 都可以无缝替换为 `Observable`。即便你现在不想要或者不需要 Reactive Extensions 的威力，它也能够在不引入无法向后兼容的变更的情况下，对你的代码实现演进。另外，真正享受 RxJava 提供的各种可能性的是客户端，比如延迟执行、声明式并发和异步链。更棒的是，由于 `Observable` 到 `BlockingObservable` 的无缝转换，传统客户端可以按照它们的需要消费 API，你可以一直提供一个简单的桥接层。

你应该对 RxJava 非常有信心了，并且已经理解在遗留系统中使用它会带来的收益。毫无疑问，使用反应式 `Observable` 更具挑战性，在一定程度上它的学习曲线也很陡峭。但是，也不能一味夸大它的优势和面对系统增长时的可能性。设想一下，如果使用 Reactive Extensions 从头到尾实现一个完整的应用程序，又会怎样？就像一个崭新的项目，我们可以控制每个 API、接口和外部系统。第 5 章将会讨论如何编写这种应用程序，以及这样的应用程序意味着什么。

第 5 章

实现完整的反应式应用程序

托马什·努尔凯维茨（Tomasz Nurkiewicz）

在 RxJava 之禅中，经常提到的一句话就是“万物皆是流”。第 4 章介绍了如何将 RxJava 应用到已有的代码库中。但是，我们很快发现，真正的反应式应用程序大多都从头到尾使用流。这种方式能够简化对系统的理解，并且保证应用程序高度一致。非阻塞应用程序借助少量的硬件就能提供很好的性能和吞吐量。通过限制线程的数量，我们能够充分利用 CPU 资源，而不会消耗千兆字节的内存。

在 Java 中，扩展性的一个限制就是其 I/O 机制。`java.io` 包设计得非常好，包含了很多小型的 `Input`/`OutputStream` 和 `Reader`/`Writer` 实现，它们互相修饰和包装，每次只添加一项功能。尽管我非常喜欢这种优雅的关注点分离，但是在 Java 中标准 I/O 完全是阻塞的，这意味着每个想通过 `Socket` 或 `File` 进行读取和写入的线程必须无限期地等待结果。更糟糕的是，由于网络速度或磁盘旋转速度缓慢，线程卡顿在 I/O 操作中很难中断。阻塞本身不是问题，一个线程被阻塞时，其他线程仍然可以与其他打开的 `Socket` 交互。但是创建和管理线程的成本很高昂，而且线程之间的切换也需要时间。Java 应用程序完全能够处理数万个并发连接，但必须非常仔细地进行设计。RxJava 与一些事件驱动的现代库协作时，这种设计工作就能大幅减少。

5.1 解决C10k问题

C10k 问题是一个研究和优化的领域，它尝试在单个商用服务器上实现 10 000 个并发连接。即便是现在，使用传统的 Java 工具集完成这一工程任务也是颇有挑战性的。有很多反应式的方式，借助它们能够很容易达成 C10k，而借助 RxJava，能够让这些方式更具可行性。

本章将会探索几种实现技术，它们能够将扩展性提升几个数量级。这些技术都是围绕反应式编程的理念构建的。如果你非常幸运，能够从头开始一个项目，那么可能会考虑将这个项目完全以反应式的方式来实现。这样的应用程序永远不会同步地等待计算或其他操作完成。为了避免阻塞，架构必须全部是**事件驱动**和**异步**的。本节将会介绍几个简单的 HTTP 服务器样例，并观察基于我们的各种设计决策，它会有什么样的行为。不得不承认，性能和可扩展性确实非常复杂。但是，借助 RxJava，这些额外的复杂性将会大幅度降低。

每个连接对应的经典线程模型都在努力解决 C10k 问题。如果有 10 000 个线程，那么将会面临如下情况。

- 为了存储栈空间，消耗数个千兆字节的 RAM。
- 给垃圾收集机制带来巨大的压力，不过栈空间是不能进行垃圾收集的（大量的 GC 根和存活对象）。
- 浪费大量的 CPU 时间只是用于切换核心以运行各种线程（上下文切换）。

在有些场景中，经典的线程 -`Socket`（thread-per-`Socket`）模型能够很好地满足要求，事实上，直到今天它在很多应用程序上都运行得非常好。但是，在达到一定级别的并发之后，线程的数量就会变得非常危险。由单个商用服务器处理 1000 个并发连接的情况并不罕见，在长期存活的 TCP/IP 连接场景中更是如此，比如带有 `Keep-Alive` 头信息的 HTTP、服务器发送事件（server-sent event）和 WebSocket。但是，不管线程正在进行计算还是等待数据的到达，每个线程都会占据一些内存（栈空间）。

在实现扩展性方面，有两种相互独立的方式：水平扩展和垂直扩展。为了处理更多的并发连接，我们可以部署更多的服务器，每个服务器管理负载的一个子集。这需要一个前端的负载均衡器，但是这样的方式也没有解决最初的 C10k 问题，即仅用一台服务器处理负载。而另一方面，垂直扩展意味着要购买更大更强的服务器。但是，由于阻塞式 I/O，与未充分利用的 CPU 相比，需要与之不相称的内存占用。即便大型的企业服务器能够处理数十万个并发连接（价格非常高昂），但是它却远远不能解决 C10M 的问题，也就是 1000 万个并发连接。这个数字并非巧合，数年之前，在一台典型的服务器上，一个经过精心设计的 Java 应用程序就达到了如此惊人的水准。

本章将会介绍实现 HTTP 服务器的各种方式。从单线程服务器，到线程池，再到完全事件驱动的架构。这种练习背后的理念是对比实现的复杂度、性能和吞吐量。最后，你将会发现，使用 RxJava 的版本相对简洁，性能也非常出色。

5.1.1　传统的基于线程的HTTP服务器

本节将会对比阻塞式服务器（即便编写得非常好）在高负载情况下的表现。我们可能都做过这个练习：基于原始 Socket 编写一个服务器。接下来，本节将会编写一个简单的 HTTP 服务器，它会对每个请求都响应 200 OK。实际上，为了简洁，样例会完全忽略这个请求的内容。

单线程的服务器

最简单的实现就是打开一个 `ServerSocket`，并在客户端连接到达之时对其进行处理。处理某个请求时，其他的请求都要排队。如下的代码片段实际上非常简单。

```
class SingleThread {

    public static final byte[] RESPONSE = (
            "HTTP/1.1 200 OK\r\n" +
                    "Content-length: 2\r\n" +
                    "\r\n" +
                    "OK").getBytes();

    public static void main(String[] args) throws IOException {
        final ServerSocket serverSocket = new ServerSocket(8080, 100);
        while (!Thread.currentThread().isInterrupted()) {
            final Socket client = serverSocket.accept();
            handle(client);
        }
    }

    private static void handle(Socket client) {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                readFullRequest(client);
                client.getOutputStream().write(RESPONSE);
            }
        } catch (Exception e) {
            e.printStackTrace();
            IOUtils.closeQuietly(client);
        }
    }

    private static void readFullRequest(Socket client) throws IOException {
        BufferedReader reader = new BufferedReader(
                new InputStreamReader(client.getInputStream()));
        String line = reader.readLine();
        while (line != null && !line.isEmpty()) {
            line = reader.readLine();
        }
    }

}
```

除了在大学里，你可能看不到类似的底层实现，但是它确实能够运行。对于每个请求，样例忽略它实际发送的内容，而是直接返回 200 OK 的响应。在浏览器中打开 `localhost:8080`，你将会看到 `OK` 文本的成功响应。这也是该类被称为 `SingleThread` 的原因。`ServerSocket.accept()` 会一直阻塞，直到与某个客户端建立连接为止。然后，它返回一个客户端 `Socket`。与这个 `Socket` 进行交互（读取或写入）时，样例依旧还在监听传入的连接，但是这些连接不会得到处理，因为线程正忙于处理第一个客户端。这就像医生的诊室一样：一个病人进入就诊时，其他的病人只能排队等待。注意到 `8080` 参数（监听端口）后面还有一个额外的参数 `100` 了吗？这个值（默认值为 `50`）限制了在队列中可以等待的挂起连接的数量。如果超过这个值，连接就会被拒绝。更糟糕的是，我们在这里假定实现的是 HTTP/1.1，它默认使用持久化连接。在客户端连接关闭之前，样例会保持 TCP/IP 连接处于打开状态，这会阻塞新的客户端。

现在，回到客户端连接上，首先读取整个请求，然后写入响应。这两个操作可能都会阻塞，并受网络缓慢或拥堵的影响。如果某个客户端建立了连接，但是等待了几秒之后才发送请求，那么其他的客户端必须要等待。让一个线程来处理所有的传入连接显然不具备扩展性，我们仅仅解决了 C1（一个并发连接）问题。

附录 A 包含了源代码，以及关于其他阻塞式服务器的讨论。本章不再花费更多的时间分析非扩展性的阻塞式架构，而是对它们进行简要地总结，这样，就能对它们进行快速的基准测试和统一的对比。

在 A.1 节，你将会看到使用 C 语言的 `fork()` 编写的简单服务器的源代码。尽管看上去非常简单，但是它为每个新的客户端连接均建立了一个新的进程，这样会给操作系统带来很沉重的负载，对于存活时间非常短的连接来说，情况会更加严重。每个进程都需要很多内存而且初始化过程会消耗一定的时间。另外，上千个进程的启动或停止总是会不必要地占用系统资源。

`ThreadPerConnection`（参见 A.2 节）展示了如何实现阻塞式服务器，它为每个客户端连接都创建了一个新线程。它的扩展性可能不错，但是会遇到与 C 语言中的 `fork()` 同样的问题：启动一个新的线程会消耗一定的时间和资源。对于短期存活的连接来说，这显得尤其浪费。除此之外，可同时运行的客户端线程并没有最大数量的限制。在计算机系统中，如果你不对某个事情进行限制，问题将会以最糟糕的方式在你最不期望的地方出现。例如，如果出现上千的并发连接，程序将会变得不稳定，并且最终会因 `OutOfMemoryError` 而崩溃。

`ThreadPool`（参见 A.3 节）同样也会为每个连接建立一个线程，但是客户端断开连接时，线程会被回收，这样，就节省了为每个客户端预热线程的成本。这与流行的 Servlet 容器（如 Tomcat 和 Jetty）的运行方式非常类似，它们会默认在池中维持 100 到 200 个线程。Tomcat 有所谓的 NIO 连接器，会以异步的方式处理 Socket 上一些操作，但是 Servlet 以及基于 Servlet 构建的框架依然会以阻塞式的方式运行。这意味着传统的应用程序即便使用现代化的 Servlet 容器，最多也只能有数千个连接。

5.1.2 借助Netty和RxNetty实现非阻塞的HTTP服务器

现在，关注一下使用事件驱动的方式来编写 HTTP 服务器，这种方式在扩展性方面更值得期待。在阻塞式的处理模型中，每个请求对应一个线程显然无法进行扩展。我们需要一种仅用少量线程就能管理大量客户端连接的方式。这种方式具有如下优点。

- 减少内存消耗。
- 更好的 CPU 和 CPU 缓存利用率。
- 在单个节点上极大地提升可扩展性。

但是这种方式会以失去简洁性和清晰性为代价。线程不允许在任何操作上阻塞，我们不能再假设通过线路接收或发送数据的线程与执行本地方法调用的线程相同。延迟是难以预测的，而响应时间也会高几个数量级。到你阅读本书的时候，可能还有很多旋转磁盘驱动器，它们甚至比局域网还要慢。本节将会使用 Netty 框架开发一个小型的事件驱动应用程序，随后将其重构为 RxNetty。最后，本节会对所有的方案进行基准测试。

Netty 完全是事件驱动的，不必阻塞等待数据发送或接收。相反，原始字节会以 ByteBuf 实例的形式推送到处理管道中。TCP/IP 给人的印象是连接和数据会在两台计算机之间一个字节接一个字节地流动。但事实上，TCP/IP 是构建在 IP 上的，IP 几乎不能传递块状的数据，也就是 packet。操作系统负责按照正确的顺序组装它们，并制造一种流的假象。Netty 放弃了这种抽象，它在字节序列层运行，而不是在流的层面。字节到达应用程序时，Netty 就会通知处理器。发送字节时，会得到一个没有阻塞的 ChannelFuture（关于 future 的更多内容会在后文阐述）。

我们的非阻塞 HTTP 服务器有三个组件。第一个组件会启动服务器并搭建环境，如下所示。

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;

class HttpTcpNettyServer {

    public static void main(String[] args) throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            new ServerBootstrap()
                    .option(ChannelOption.SO_BACKLOG, 50_000)
                    .group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new HttpInitializer())
                    .bind(8080)
                    .sync()
                    .channel()
                    .closeFuture()
                    .sync();
        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

这是使用 Netty 编写的最基础的 HTTP 服务器。核心部分是负责接收传入连接的 bossGroup 池和处理事件的 workerGroup。这些池都不大：bossGroup 大小为 1，而 workerGroup 则应该接近 CPU 核心的数量。对于编写良好的 Netty 服务器来说，这已经足够了。在这里，除了监听 8080 端口之外，我们还没有指定服务器应该做什么。具体做什么可以通过 ChannelInitializer 来配置，如下所示。

```
import io.netty.channel.ChannelInitializer;
import io.netty.channel.socket.SocketChannel;
import io.netty.handler.codec.http.HttpServerCodec;

class HttpInitializer extends ChannelInitializer<SocketChannel> {

    private final HttpHandler httpHandler = new HttpHandler();
```

```
    @Override
    public void initChannel(SocketChannel ch) {
        ch
                .pipeline()
                .addLast(new HttpServerCodec())
                .addLast(httpHandler);
    }
}
```

这里并不是提供了一个处理连接的函数，而是构建了一个管道，处理传入的 ByteBuf 实例。这个管道的第一步是将传入的原始字节解码为更高层级的 HTTP 请求对象。这个处理器是内置的。它还能够将 HTTP 响应编码为原始字节。在更为健壮的应用程序中，你通常会看到更多的处理器聚焦在更小的功能上，例如，帧解码、协议解码、安全性等。每段数据和通知都会流经该管道。

你可能已经看到与 RxJava 类似的地方了。管道的第二步是业务逻辑组件，它会实际处理请求，而不仅仅是对其进行拦截或增强。尽管 HttpServerCodec 在本质上是有状态的（它将传入的包转换成了更高层级的 HttpRequest 实例），但是自定义的 HttpHandler 可以是一个无状态的单例类。

```
import io.netty.channel.*;
import io.netty.handler.codec.http.*;

@Sharable
class HttpHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.flush();
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        if (msg instanceof HttpRequest) {
            sendResponse(ctx);
        }
    }

    private void sendResponse(ChannelHandlerContext ctx) {
        final DefaultFullHttpResponse response = new DefaultFullHttpResponse(
                HTTP_1_1,
                HttpResponseStatus.OK,
                Unpooled.wrappedBuffer("OK".getBytes(UTF_8)));
        response.headers().add("Content-length", 2);
        ctx.writeAndFlush(response);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        log.error("Error", cause);
        ctx.close();
    }
}
```

在构建完响应对象之后，样例通过 write() 将 DefaultFullHttpResponse 写回。但是，write() 与常规的 socket 并不相同，它并不会阻塞。相反，它会返回一个 ChannelFuture，我们可以通过 addListener() 对其进行订阅并异步关闭通道，如下所示。

```
ctx
    .writeAndFlush(response)
    .addListener(ChannelFutureListener.CLOSE);
```

通道（channel）是对通信连接的抽象，例如 HTTP 连接，因此关闭通道就会关闭连接。为了实现持久化连接，我们不希望这样做。

Netty 只用屈指可数的线程就能处理上百个连接。我们没有为每个连接保留任何重量级的数据结构或线程，这与本质更加接近。计算机接收到一个 IP 包，并唤醒监听该目标端口的进程。TCP/IP 连接通常会使用线程来实现抽象，但是，如果应用程序需要更高的负载和连接数，直接在包级别进行操作会更加可靠。我们依然会有通道（线程的轻量级表述）和管道的概念，以及可能会有状态的处理器。

使用RxNetty实现Observable服务器

Netty 是很多成功产品和框架的重要支撑，比如 Akka、Elasticsearch、HornetQ、Play 框架、Ratpack 和 Vert.x。围绕 Netty 也有很薄的一层抽象，通过这层抽象能够将 Netty API 和 RxJava 连接在一起。接下来，让我们使用 RxNetty 重写非阻塞的 Netty 服务器。先从一个简单的货币服务器入手来熟悉 API，如下所示。

```
import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;
import io.reactivex.netty.protocol.tcp.server.TcpServer;

class EurUsdCurrencyTcpServer {

    private static final BigDecimal RATE = new BigDecimal("1.06448");

    public static void main(final String[] args) {
        TcpServer
            .newServer(8080)
            .<String, String>pipelineConfigurator(pipeline -> {
                pipeline.addLast(new LineBasedFrameDecoder(1024));
                pipeline.addLast(new StringDecoder(UTF_8));
            })
            .start(connection -> {
                Observable<String> output = connection
                    .getInput()
                    .map(BigDecimal::new)
                    .flatMap(eur -> eurToUsd(eur));
                return connection.writeAndFlushOnEach(output);
            })
            .awaitShutdown();
    }

    static Observable<String> eurToUsd(BigDecimal eur) {
        return Observable
            .just(eur.multiply(RATE))
```

```
                .map(amount -> eur + " EUR is " + amount + " USD\n")
                .delay(1, TimeUnit.SECONDS);
    }
}
```

这是一个基于 RxNetty 编写的自给自足的 TCP/IP 服务器。你应该对它的主要组成部分有一个大致的了解。首先，我们创建了一个新的 TCP/IP 服务器来监听 8080 端口。Netty 为流经管道的 ByteBuf 消息提供了非常低层级的抽象。样例必须要配置这样的管道。第一个处理器会使用内置的 LineBasedFrameDecoder 将 ByteBuf 序列重新排列（按需进行拆分和连接）为行数据的序列。随后，解码器会将包含所有数据行的 ByteBuf 转换为 String 实例。从此处开始，就可以只处理 String 了。

每次新的连接到达，回调就会执行。Connection 对象允许异步发送和接收数据。首先从 connection.getInput() 开始。这个对象是 Observable<String> 类型的，每次客户端新的一行请求在服务器出现，它就会发布一个值。getInput()Observable 会以异步的方式通知新的输入。首先将 String 解析为 BigDecimal，然后使用辅助方法 eurToUsd()，假装调出一些货币兑换服务。为了让这个样例看上去更加真实，我们还人为地应用了 delay()，这样，必须等待一会儿才能得到响应。显然，delay() 是异步的，不会涉及任何的休眠。与此同时，我们会不断地接收和转换请求。

在将所有的转换完成后，output Observable 会直接输入到 writeAndFlushOnEach() 方法。这些内容很容易理解，接收一个输入序列，对它们进行转换，然后将转换后的序列作为输出。现在，使用 telnet 与服务器进行交互。请注意一下，有些响应是在消费多个请求之后才出现的，这是伪造的货币服务器存在延迟的缘故。

```
$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
2.5
2.5 EUR is 2.661200 USD
0.99
0.99 EUR is 1.0538352 USD
0.94
0.94 EUR is 1.0006112 USD
20
30
40
20 EUR is 21.28960 USD
30 EUR is 31.93440 USD
40 EUR is 42.57920 USD
```

服务器被视为将请求数据转换为响应数据的函数。因为 TCP/IP 不仅仅是一个简单的函数，有时候还是由相互依赖的数据块组成的流，在这种场景下 RxJava 运行得也非常好。借助一组非常丰富的操作符，能够很容易地将输入转换为输出。当然，输出流并不一定是要基于输入的。例如，如果你实现服务器端推送事件，服务器端只是发布数据而已，与传入的数据并没有关联。

EurUsdCurrencyTcpServer 是反应式的，因为只有数据传入的时候，它才会采取相应的行

为。对于每个客户端，我们不会为其创建单独的线程。这种实现方式可以很容易地承担数千个并发连接，而且垂直可扩展性仅会受到必须要处理的通信量的限制，而不会受到或多或少空闲连接数量的限制。

现在，我们已经知道了 RxNetty 的工作原理，接下来可以回到返回 OK 响应的原始 HTTP 服务器。RxNetty 内置了对 HTTP 客户端和服务器端的支持，但是我们将从基于 TCP/IP 的简单实现开始进行讲解。

```
import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;
import io.reactivex.netty.examples.AbstractServerExample;
import io.reactivex.netty.protocol.tcp.server.TcpServer;

import static java.nio.charset.StandardCharsets.UTF_8;

class HttpTcpRxNettyServer {

    public static final Observable<String> RESPONSE = Observable.just(
            "HTTP/1.1 200 OK\r\n" +
            "Content-length: 2\r\n" +
            "\r\n" +
            "OK");

    public static void main(final String[] args) {
        TcpServer
            .newServer(8080)
            .<String, String>pipelineConfigurator(pipeline -> {
                pipeline.addLast(new LineBasedFrameDecoder(128));
                pipeline.addLast(new StringDecoder(UTF_8));
            })
            .start(connection -> {
                Observable<String> output = connection
                    .getInput()
                    .flatMap(line -> {
                        if (line.isEmpty()) {
                            return RESPONSE;
                        } else {
                            return Observable.empty();
                        }
                    });
                return connection.writeAndFlushOnEach(output);
            })
            .awaitShutdown();
    }
}
```

掌握了 EurUsdCurrencyTcpServer 之后，再去理解 HttpTcpRxNettyServer 就应该非常容易了。出于教学的目的，样例始终返回静态的 200 OK 响应，在这里解析请求没有太大的意义。但是，设计良好的服务器在读取请求之前是不应该发送响应的。因此，首先在 getInput() 中查找一个空行，它代表了 HTTP 请求的结束。在此之后，才生成 200 OK 的响应行。按照这种方式构建的 output Observable 会被传递给 connection.writeString()。换句话说，一旦遇到请求包含空行的场景，响应将会立即发送给客户端。

使用 TCP/IP 来实现 HTTP 服务器只是一个练习而已，它能够帮助你理解 HTTP 的复杂之处。幸好我们不必每次都使用 TCP/IP 抽象来实现 HTTP 和 RESTful Web 服务。与 Netty 类似，RxNetty 也有一些用于 HTTP 功能的内置组件，如下所示。

```
import io.reactivex.netty.protocol.http.server.HttpServer;

class RxNettyHttpServer {

    private static final Observable<String> RESPONSE_OK =
        Observable.just("OK");

    public static void main(String[] args) {
        HttpServer
            .newServer(8086)
            .start((req, resp) ->
                resp
                    .setHeader(CONTENT_LENGTH, 2)
                    .writeStringAndFlushOnEach(RESPONSE_OK)
            ).awaitShutdown();
    }

}
```

如果你厌倦了只返回静态的 `200 OK`，那么我们可以相对轻松地构建非阻塞的 RESTful Web 服务，同样适用于实现货币交换功能。

```
class RestCurrencyServer {

    private static final BigDecimal RATE = new BigDecimal("1.06448");

    public static void main(final String[] args) {
        HttpServer
                .newServer(8080)
                .start((req, resp) -> {
                    String amountStr = req.getDecodedPath().substring(1);
                    BigDecimal amount = new BigDecimal(amountStr);
                    Observable<String> response = Observable
                            .just(amount)
                            .map(eur -> eur.multiply(RATE))
                            .map(usd ->
                                    "{\"EUR\": " + amount + ", " +
                                     "\"USD\": " + usd + "}");
                    return resp.writeString(response);
                })
                .awaitShutdown();
    }
}
```

可以使用 Web 浏览器或 `curl` 与这个服务器进行交互。为了去除请求中的第一条斜线，这里用到了 `substring(1)`。

```
$ curl -v localhost:8080/10.99

> GET /10.99 HTTP/1.1
```

```
> User-Agent: curl/7.35.0
> Host: localhost:8080
> Accept: */*
>

< HTTP/1.1 200 OK
< transfer-encoding: chunked
<

{"EUR": 10.99, "USD": 11.6986352}
```

现在有了这个简单 HTTP 服务器的多个实现，我们可以对比它们的性能、可扩展性和吞吐量。这也是放弃熟悉的基于线程的模型，转而着手使用 RxJava 和异步 API 的原因。

5.1.3　阻塞式和反应式服务器的基准测试

为了阐述非阻塞、反应式 HTTP 服务器的价值和成功的原因，本节将为每种实现方式运行一系列基准测试。有意思的是，我们选择的基准测试工具 `wrk` 也是非阻塞的。否则，它本身无法模拟数万个并发连接的负载。另外一个可选的工具是 Gatling，它构建在 Akka 工具集之上。传统的基于线程的工具无法模拟这种大量的负载，如 JMeter 和 `ab`，它们本身就会成为瓶颈。

每个基于 JVM 的实现[1]都会针对 10 000、20 000 和 50 000 个并发 HTTP 客户端（也就是 TCP/IP 连接）进行基准测试。令人感兴趣的是每秒的请求数量（吞吐量），以及响应时间的中位数和第 99 个百分位数。需要提醒一下：中位数意味着 50% 的请求响应速度满足给定值，而第 99 个百分位数意味着 1% 的请求要比给定的值更慢。

基准环境

所有的基准测试都运行在基于 Linux 3.13.0-62-generic 核心家用笔记本电脑上，硬件配置为 Intel i7 CPU 2.4 GHz，8 GB RAM 和固态硬盘驱动器（SSD）。客户端机器运行官网版本的 `wrk`，Gatling 和 JMeter 工具通过千兆以太路由器连接至服务器端机器。客户端和服务器端机器之间运行 `ping` 的平均时间为 289 μs（偏差 42 μs，最低 160 μs）。

每项基准测试都会至少运行一分钟，包括 30 秒对 JDK 1.8.0_66 的预热。使用的软件版本为 RxJava 1.0.14、RxNetty 0.5.1 和 Netty 4.0.32.Final。软件负载通过 `htop` 进行测量。

基准测试是通过下面的命令执行的（使用 `-c` 参数表示并发客户端的数量）。

```
wrk -t6 -c10000 -d60s --timeout 10s --latency http://server:8080
```

1. 返回200 OK的简单服务器

第一个基准测试对比了各种实现方式只返回简单的 200 OK 而不执行任何后台任务时的性能。这个基准测试可能不太符合现实，但是它能够让我们对服务器和以太网的上限有一个

注 1：不包括 C 语言和其他反应式平台，如 Node.js。

基本的了解。在后续的测试中，将会为每个服务器添加任意的休眠时间。

图 5-1 展现了各个实现方式每秒的请求数（注意对数尺度）。

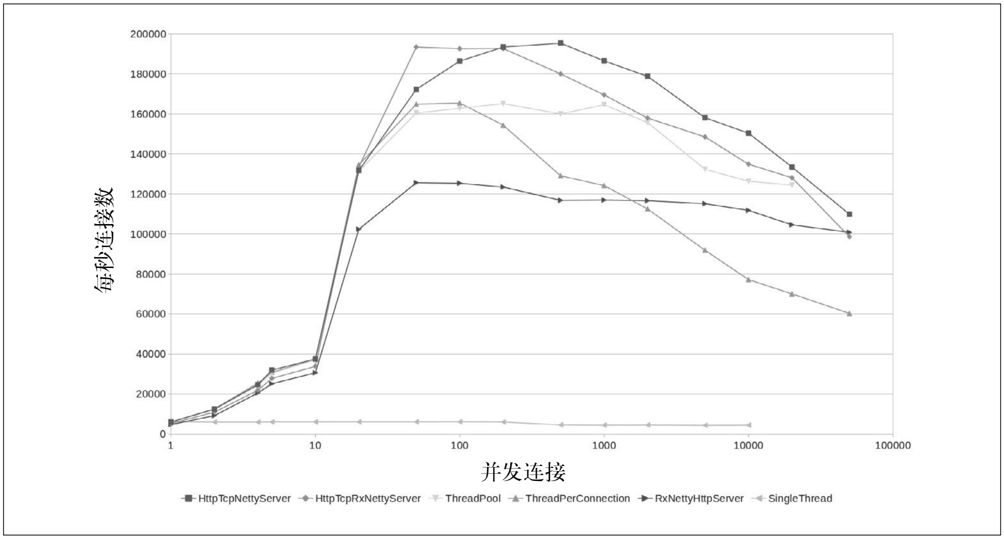


图 5-1

注意，这个基准测试仅仅是一个热身，随后服务器端会运行一些工作内容。但是，从这里，我们已经能够看出一些有意思的趋势。

- 基于 Netty 和 RxNetty 的实现使用了原始的 TCP/IP，吞吐量最佳，几乎达到了每秒 200 000 个请求。
- 不出所料的是，`SingleThread` 要慢得多，无论并发级别如何，它每秒大约只能处理 6000 个请求。
- 但是，在只有一个客户端的时候，`SingleThread` 是最快的实现方式。线程池、事件驱动的（Rx）Netty 以及其他实现方式的开销是显而易见的。随着客户端数量的增加，这种优势迅速消耗殆尽。此外，服务器的吞吐量还高度依赖于客户端的性能。
- 令人稍感意外的是，`ThreadPool` 的运行效果非常好，但是在高负载的情况下会变得不稳定(`wrk` 报告了很多错误)。遇到 50 000 个并发连接时,就会完全失败了(达到 10 秒超时)。
- `ThreadPerConnection` 开始运行得也非常好，但是超过 100~200 个线程时，服务器的吞吐量会快速降低。同时，50 000 个线程会给 JVM 带来很大的压力，尤其是令人头疼的额外千兆字节的栈空间。

我们不再花费更多时间来分析这个有点牵强的基准测试，毕竟服务器很少直接返回一个响应。因此，我们想要模拟一下针对每个请求都执行一定的工作内容的场景。

2. 模拟服务器端的工作

为了模拟服务器端的工作，样例会在请求和响应之间注入对 `sleep()` 的调用。这样做也是有一定道理的：服务器在响应用户请求的时候，通常并不会执行任何 CPU 密集型的任务。传统的服务器针对每个请求会使用一个线程，并在外部资源上阻塞。而反应式服务器只需

等待一个外部信号（如包含响应的事件或消息），同时释放底层的资源。

鉴于此，对于阻塞式的实现，样例只需要添加 sleep() 即可；而对于非阻塞的服务器，我们会使用 Observable.delay() 或类似的做法，以便模拟调用非阻塞、慢响应的外部服务，如下所示。

```
public static final Observable<String> RESPONSE = Observable.just(
        "HTTP/1.1 200 OK\r\n" +
        "Content-length: 2\r\n" +
        "\r\n" +
        "OK")
    .delay(100, MILLISECONDS);
```

在阻塞式实现中使用非阻塞延迟并没有太大意义，因为即便底层实现是非阻塞的，服务器依然需要等待响应。也就是说，如果为每个请求注入了 100 毫秒的延迟，那么每次与服务器的交互至少需要十分之一秒。现在的基准测试更加符合现实，也更有意思。每秒请求数与客户端连接的关系如图 5-2 所示。

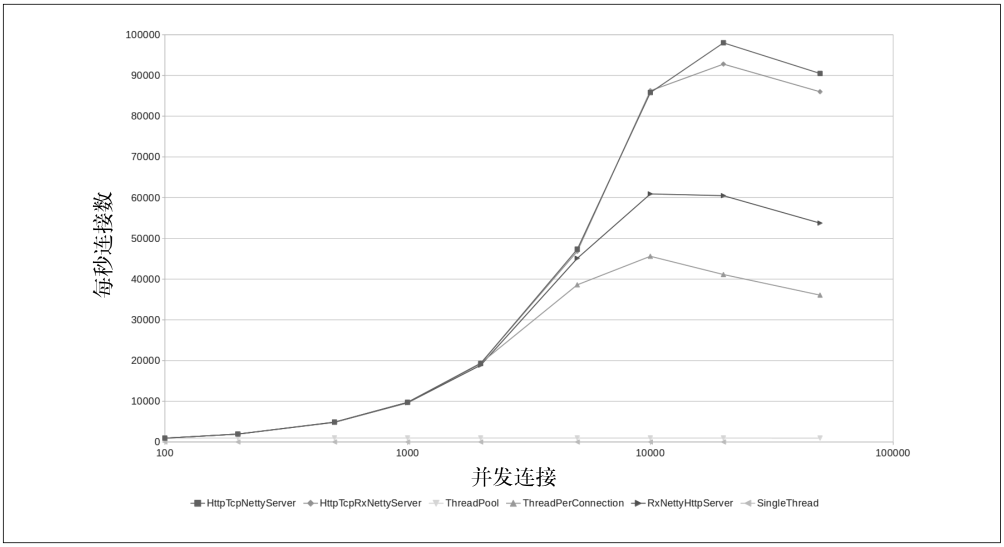


图 5-2

这个结果更符合对真实负载的预期。图的顶部是两个基于 Netty 的实现（HttpTcpNetty-Server 和 HttpTcpRxNettyServer），它们目前是最快的，轻松实现每秒 90 000 个请求（Request Per Second，RPS）。实际上，在 10 000 个并发客户端之前，服务器是线性扩展的。这一点很容易证明：每个客户端大约生成 10 RPS（每个请求大约消耗 100 毫秒，所以 1 秒内可以发送 10 个请求）。两个客户端会生成 20 RPS，5 个客户端会生成 50 RPS，以此类推。在 10 000 个并发连接的时候，预期的结果是 100 000 RPS，实际上已经接近理论极限了（90 000 RPS）。

在底部，我们看到的是 SingleThread 和 ThreadPool 服务器。它们的性能结果非常糟糕，这一点并不令人意外。SingleThread 只有一个线程处理请求，而每个请求至少要消耗 100

毫秒，那么其处理能力显然不能超过 10 RPS。ThreadPool 要好得多，它有 100 个线程，每个线程的处理能力是 10 RPS，总数能够达到 1000 RPS。与反应式 Netty 和 RxJava 实现相比，这个数据差了好几个数量级。另外，在高负载的情况下，SingleThread 几乎拒绝了所有的请求。在大约 50 000 个并发连接的时候，它只能接收少量的请求，而且几乎无法满足 wrk 要求的 10 秒超时的限制。

你可能会问，为什么要限制 ThreadPool 只有 100 个线程呢？因为这个数字与流行的 HTTP Servlet 容器的默认值是比较接近的，当然样例可以指定更大的值。连接都是持久的，在整个连接期间，它会一直占用池中的线程，所以可以将 ThreadPerConnection 视为没有任何线程数量限制的线程池。令人意外的是，这种实现方式运行得非常好，即便是在 JVM 管理 50 000 个并发线程（每个线程对应一个连接）的时候也是如此。实际上，ThreadPerConnection 并没有比 RxNettyHttpServer 差太多。事实证明，仅仅通过 RPS 来衡量吞吐量还是不够的，样例还必须要查看每个请求的响应时间。这取决于需求，但是一般情况下，你既需要高吞吐以便于充分利用服务器，又需要低延迟为用户提供**良好**的性能体验。

平均响应时间是一个很好的指标。一方面，平均值会隐藏异常值（一些慢得令人难以接受的请求）；另一方面，**典型**响应时间（大多数客户端观察到的值）会小于平均值，这同样是那些异常值导致的。事实证明，百分位数会更具有代表性，它能够有效描述特定值的分布情况。图 5-3 展现了每种服务器实现响应时间的第 99 个百分位数与并发连接（或客户端）数量之间的关系。*Y* 轴的值代表 99% 的请求都比给定的值更快。显然，这些值越小越好（但是不可能低于模拟延迟的 100 毫秒），并且随着负载的增加，它们增长得越少越好。

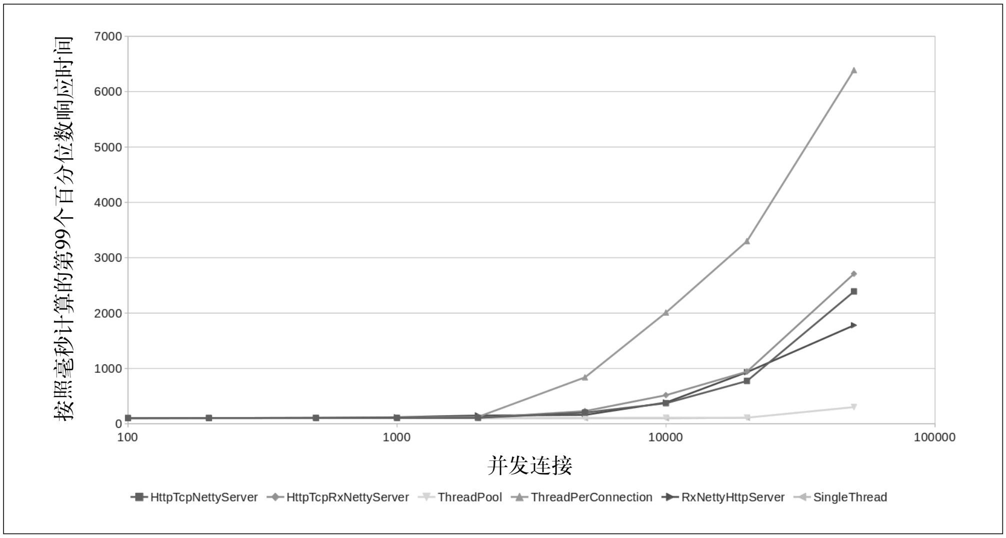


图 5-3

ThreadPerConnection 实现方式非常突出。在 1000 个并发连接之前，所有实现方式的表现不分上下。但是，从某个位置开始，ThreadPerConnection 的响应就变得非常慢了，响应时间是其他竞争对手的几倍。这种现象的原因主要有两点：首先，上千个线程会有过量的上

下文切换；其次，这种实现方式的垃圾收集频率会更高。JVM 花费了太多的时间进行内部处理，留给实际工作的时间非常少。上千个线程处于空闲等待执行的状态。

你可能会惊讶于 ThreadPool 实现在响应时间的第 99 个百分位数上的突出表现。它优于其他所有的实现方式，并且在高负载的情况下依然能够保持稳定。让我们简单看一下 ThreadPoold 实现方式大致的样子，如下所示。

```
BlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(1000);
executor = new ThreadPoolExecutor(100, 100, 0L, MILLISECONDS, workQueue,
        (r, ex) -> {
            ((ClientConnection) r).serviceUnavailable();
        });
```

在这里没有使用 Executors 生成器（builder），而是直接构造了 ThreadPoolExecutor，从而完全控制 workQueue 和 RejectedExecutionHandler。当 workQueue 空间不足时，则会运行 RejectedExecutionHandler。在这里，为了防止服务器出现过载，无法得到快速处理的请求会立即被拒绝。其他实现方式均没有这种安全特性，它被称为**快速失败**（fail-fast）。8.2 节将会简要介绍快速失败的功能。ThreadPool 方案的响应性与其暴露的错误率的关系，如图 5-4 所示。

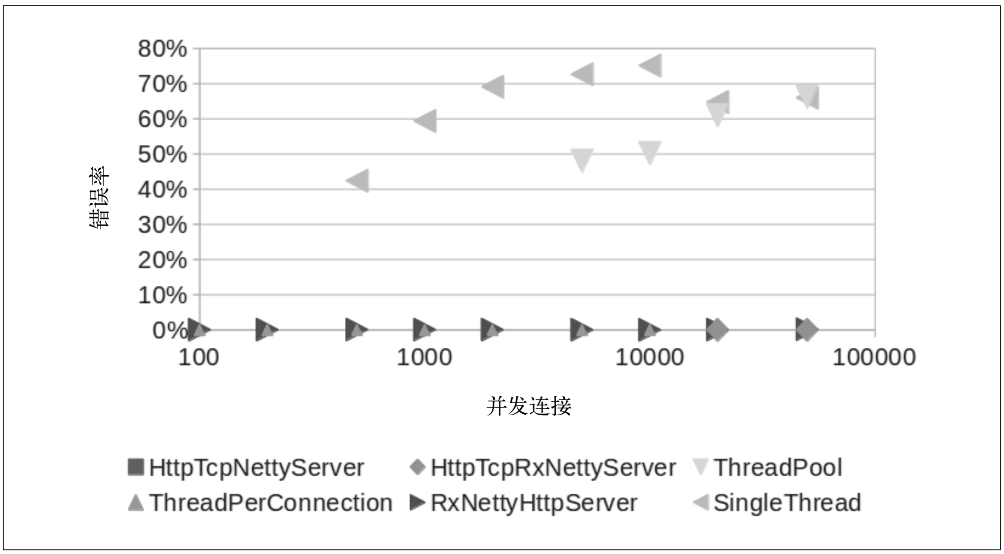


图 5-4

除了 SingleThread 和 ThreadPool 实现，wrk 负载测试工具对其他实现均没有报告错误。这是一种很有意思的权衡：ThreadPool 总是能够尽快响应，比其他的竞争者要快得多，但是，在它出现超载的时候，也会立即拒绝请求。当然，你也可以基于 Netty/RxJava 的反应式实现执行类似的机制。

总而言之，采用线程池和独立线程的方案无法满足吞吐量和响应时间的需求，而这些需求可以通过反应式实现来满足。

5.1.4 反应式HTTP服务器之旅

TCP/IP 以及基于它构建的 HTTP 本质上都是事件驱动的。尽管它提供了输入和输出管道的错觉，但是在底层可以看到异步数据包异步到达。与计算机科学领域的其他抽象类似，将网络栈视为字节组成的阻塞流是有问题的，想要充分利用硬件时更是如此。

即便是在中等负载的情况下，采用传统的网络方式依然是可行的。但是，如果要超越传统 Java 应用程序的极限，就必须要采用反应式的方法了。尽管 Netty 是构建反应式、事件驱动的网络应用程序的优秀框架，但是很少直接使用。相反，它会作为各种库和框架的一部分，包括 RxNetty。RxNetty 非常有意思，它将事件驱动网络的优势与 RxJava 操作符的简洁性组合在了一起。我们依然会将网络通信视为信息（包）组成的流，不过将其抽象成了 `Observable<ByteBuf>`。

还记得 5.1 节是如何定义 10 000 个并发连接问题的吗？我们使用各种 Netty 和 RxNetty 实现成功解决了这个问题。实际上，成功实现的服务器能够经受 C50k 的考验，也就是处理 50 000 个并发 HTTP 持久化连接。如果有更多的客户端硬件（因为服务器端运行得很好）和更低的请求通过线路频率，相同的实现可以很轻松地经受住 C100k 甚至更高的并发连接，而实现这一点，只用了十几行代码。

显然，实现 HTTP（或其他任何协议，这里以 HTTP 为例仅仅是因为它的普遍性）的服务器部分只是整个过程的一个方面，同样重要的是服务器正在做的事情。大多数情况下，HTTP 服务器会成为其他服务器的客户端。到目前为止，本章关注的是反应式、非阻塞的 HTTP 服务器。这样做是合理的，但是阻塞式代码会在很多出乎意料的地方潜入。我们过度关注服务器端，而完全忽略了客户端。但是现代服务器也会扮演客户端的角色，尤其是在分布式系统中，它们会请求数据并将其推送至下游服务。可以大胆地假设，对流行搜索引擎的一次请求可能会跨越数百甚至数千个下游组件，从而产生大量的客户端请求。显然，如果这些请求是阻塞式和序列化的，那么搜索引擎的响应时间将会非常长。

不管服务器基础设施的代码实现得多好，如果它依然要处理阻塞式 API，可扩展性就会受到影响，就像基准测试显示的那样。特别是在 Java 生态系统中有一些已知的阻塞源，本书会进行简单地介绍。

5.2 HTTP客户端代码

向下游服务发送多个请求并将响应组合在一起的服务器并不少见。实际上，非常多的初创企业能够很巧妙地整合多个可用的数据源，并基于它们提供有价值的服务。如今的 API 大多都是 RESTful 风格的，SOAP 风格的 API 在不断减少，但它们都是基于 HTTP 协议的。

一个阻塞式请求就可能导致服务器停机，严重降低服务器的性能。幸好，现在有很多成熟的非阻塞式 HTTP 客户端，之前见过的 Netty 就是其中一个。非阻塞的 HTTP 客户端会试图解决两类问题，如下所示。

- 大量的独立并发请求，每个请求都需要对第三方 API 发起多轮客户端调用。这是面向服务架构的典型现象，也就是单个请求需要跨多个服务。

- 服务器发起大量的 HTTP 客户端请求，可能在批处理操作。考虑下 Web 爬虫或索引服务，它们会持续地打开数千个连接。

不管服务器的特点是什么，它们的问题都是相同的：维持大量（数万甚至更多）处于打开状态的 HTTP 连接会带来非常大的开销。如果连接的服务非常慢（此时服务器是客户端的身份），问题会更加严重，因此这种情况下需要长期持有资源。

与之相反，TCP/IP 连接是非常轻量级的。对于每个处于打开状态的连接，操作系统必须保持一个 Socket 描述符（大约 1 KB），仅此而已。数据包（消息）抵达时，内核会将它分发给对应的进程，比如 JVM。每个线程会被阻塞在一个 Socket 中，这个线程栈会占用大约 1 MB 的空间，相比之下 1 KB 就是非常小的内存占用了。换句话说，在高性能服务器上，传统的每个连接对应一个线程的模型并不能很好地扩展，需要采用底层的网络模型，而不是使用阻塞式代码对其进行抽象。RxJava + Netty 正好提供了更好的抽象，而且相对接近底层。

使用RxNetty实现非阻塞的HTTP客户端

RxJava 结合 Netty 提供了一种抽象机制，这种抽象机制非常接近网络的运行方式。它没有将 HTTP 请求视为 JVM 中的普通方法调用，而是采用了异步。另外，我们也不能再将 HTTP 视为简单的请求 – 响应协议。服务器端发送事件（server-sent event，一个请求，多个响应）、WebSockets（全双工通信）以及最终出现的 HTTP/2（在同一个线路上进行多个并行的请求和响应，彼此交织）展现了 HTTP 的各种使用场景。

在客户端，RxNetty 为简单的使用场景提供了非常简洁的 API。你可以通过组合式 `Observable` 发起请求并得到响应，如下所示。

```
Observable<ByteBuf> response = HttpClient
        .newClient("example.com", 80)
        .createGet("/")
        .flatMap(HttpClientResponse::getContent);
response
        .map(bb -> bb.toString(UTF_8))
        .subscribe(System.out::println);
```

调用 `createGet()` 方法将会返回 `Observable<HttpClientResponse>` 的一个子类。显然，客户端不会阻塞等待响应，所以 `Observable` 看上去是一个很好的选择。但这只是开始。`HttpClientResponse` 有一个 `getContent()` 方法，该方法会返回 `Observable<ByteByf>`。回忆一下 5.1.2 节，里面提到过 `ByteBuf` 是对线路接收数据块的抽象。从客户端的角度来说，这是响应的一**部分**。这没有什么问题，但是 RxNetty 比其他非阻塞 HTTP 客户端更进一步。在整个响应到达的时候，它不会简单地发出通知，相反，我们会得到一个 `ByteBuf` 消息的流，随后是一个可选的 `Observable` 完成通知。服务器端决定放弃连接的时候，我们就会得到 `Observable` 完成通知。

这样的模型非常接近 TCP/IP 栈的运行方式，并且在用例中能够更好地扩展。它能够与简单的请求 / 响应流协作，也能够用于复杂的流场景。但是，需要注意，即便是在单个响应的情况下，比如包含 HTML 的请求，它也很可能以多个数据块的方式抵达。当然，RxJava

有多种方式将它们组装回来，比如 Observable.toList() 或 Observable.reduce()。这取决于你：如果你想要在数据抵达的时候，以小数据段的方式进行消费，这完全没有问题。在这种情况下，RxNetty 的抽象级别是比较低的，但是这种抽象没有引入大的性能瓶颈，比如过多的缓冲或阻塞，所以扩展性会非常好。如果你想要使用健壮而且更高级的反应式 HTTP 客户端，可以参考 8.1.2 节。

与基于回调的反应式 API 不同，RxNetty 能够很好地与其他 Observable 协作，你可以很方便地对工作进行并行处理、组合和切分。例如，假设现在有一个 URL 的流，必须实时连接并消费数据。这个流可能是固定的（从简单的 List<URL> 构建而来），也可能是动态的，也就是新的 URL 会随时出现。如果你想要一个稳定的由数据包组成的流，并且要访问所有的资源，那么可以对它们进行 flatMap() 操作。

```
Observable<URL> sources = //...

Observable<ByteBuf> packets =
    sources
    .flatMap(url -> HttpClient
        .newClient(url.getHost(), url.getPort())
        .createGet(url.getPath()))
    .flatMap(HttpClientResponse::getContent);
```

这个例子稍微有些牵强，它将来自不同源的 ByteBuf 消息混合在了一起，但是你要掌握其中的思想。对于上游 Observable 中的每个 URL，都会根据该 URL 生成一个由 ByteByf 实例组成的异步流。如果你想要先转换传入的数据，可能需要将数据块组合成一个事件，这很容易实现，比如通过 reduce()。最终的结果就是这样：你可以轻松拥有数万个处于打开状态的 HTTP 连接，它们要么处于空闲状态，要么在接收数据。这里的限制因素并不是内存，而是 CPU 的处理能力以及网络带宽。如果要处理合理数量范围内的事务，JVM 并不需要 GB 级别的内存消耗。

在现代应用程序中，HTTP 的 API 是一个重要的瓶颈。在 CPU 方面，它们并不昂贵，但是阻塞式的 HTTP 就像普通的方法调用一样，大大限制了扩展性。即便你小心翼翼地移除了阻塞式的 HTTP 通信，同步代码还是会在意想不到的地方出现。java.net.URL 的 equals() 方法有一个缺陷，就是它会发起网络调用。当你对比 URL 类的两个实例时，这个看似很快的方法其实会发起一轮网络的往返（调用序列，自上而下读取）。

```
java.net.URL.equals(URL.java)
java.net.URLStreamHandler.equals(URLStreamHandler.java)
java.net.URLStreamHandler.sameFile(URLStreamHandler.java)
java.net.URLStreamHandler.hostsEqual(URLStreamHandler.java)
java.net.URLStreamHandler.getHostAddress(URLStreamHandler.java)
java.net.InetAddress.getByName(InetAddress.java)
java.net.InetAddress.getAllByName(InetAddress.java)
java.net.InetAddress.getAllByName0(InetAddress.java)
java.net.InetAddress.getAddressesFromNameService(InetAddress.java)
java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java)
[native code]
```

为了判断两个 URL 是否相等，JVM 会调用 lookupAllHostAddr()，它（在 native 代码中）会调用 gethostbyname（或类似的方法），向 DNS 服务器发起一次同步请求。如果你的线

程数量有限，并且其中有一部分意外阻塞时，这可能会造成灾难性的影响。还记得基于 RxNetty 的服务器吗？它们最多只会使用几十个线程。另外一种灾难性的场景就是频繁调用 `URL.equals()` 的时候，比如在 `Set<URL>` 中。`URL` 这种意料之外的行为是众所周知的，这就好比一个事实：它的 `equals()` 实际会生成的结果取决于 Internet 的连接状况。

指出这一点只是为了说明编写反应式应用程序非常困难，而且陷阱重重。下一节将会介绍另外一个更明显的阻塞源：数据库访问代码。

5.3 关系数据库访问

在前面的部分，我们已经得出结论：每个服务器最终都会成为不同服务的客户端。还有一个非常有意思的现象，目前使用的计算机系统几乎全都是分布式的。由网线连接的两台计算机需要进行通信时，在空间上实际已经是分布式的了。更极端一点，你可以将每台计算机都想象成一个分布式系统，各个独立的 CPU 核心缓存并不总是一致的，它们必须通过消息传递协议来实现同步。现在，让我们对比一下应用程序服务器和数据库服务器架构。

在 Java 领域，关系数据库访问长期存在的标准是 **Java 数据库连接**（Java Database Connectivity，JDBC）。从消费者的角度来说，JDBC 提供了一组 API 与关系数据库进行通信，这些数据库包括 PostgreSQL、Oracle Database 等。核心的抽象是 `Connection`（TCP/IP，线路连接）、`Statement`（数据库查询）和 `ResultSet`（查看数据库结果）。如今，开发人员很少会直接使用这个 API，因为已经有了更易于使用的抽象，从 Spring 框架中轻量级的 `JdbcTemplate`，到像 jOOQ 这样的代码生成库，再到像 JPA 这样的对象 – 关系映射解决方案。JDBC 在错误处理与检查型异常（自从 Java 7 提供了 try-with-resources 后已经简单了很多）方面饱受争议。

```
import java.sql.*;

try (
        Connection conn = DriverManager.getConnection("jdbc:h2:mem:");
        Statement stat = conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT 2 + 2 AS total")
) {
    if (rs.next()) {
        System.out.println(rs.getInt("total"));
        assert rs.getInt("total") == 4;
    }
}
```

上述的样例使用了嵌入式的 H2 数据库，这个数据库通常用来进行集成测试。但是在生产环境中，数据库实例和应用程序很少运行在同一台机器上。每次与数据库的交互都需要一次网络往返。JDBC 的核心是它的 API，每个数据库厂商都要实现该 API。

请求 JDBC 的 API 获取新的 `Connection` 时，API 的实现必须要发起一个到数据库的物理连接，这涉及打开客户端 Socket、授权等操作。数据库有不同的线路协议（几乎都是二进制的），JDBC 实现（也被称为 `Driver`）的责任就是将底层的网络协议转换为一致的 API。这种方式运行得非常好（抛开不同的 SQL 方言不谈），但是在 1997 年发布 JDBC 标准和 JDK 1.1 的时候，没有人预料到 20 年后反应式和异步编程会变得如此重要。当然，API 经历了

很多的版本，但是都是固有阻塞的，等待每个数据库操作完成。

这其实和之前讨论的 HTTP 问题相同。应用程序中必须包含与活跃数据库操作（查询）数量相同的线程。JDBC 是以可移植的方式访问各种关系数据库的唯一成熟标准（再次强调，暂时不考虑不同 SQL 方言的差异）。几年前，Servlet 规范在 3.0 版本有了显著的改进，引入了 `HttpServletRequest.startAsync()` 方法。但非常糟糕的是，JDBC 标准依然固守着经典的模型。

JDBC 保持阻塞有多重原因。Web 服务器可以很轻松地处理数十万个处于打开状态的连接。例如，如果 HTTP 连接上只是偶尔流动一小段数据。另一方面，数据库系统会对每个客户端查询执行几个类似的步骤，如下所示。

(1) **查询解析**（CPU 密集）：将一个包含查询的 `String` 转换为一个解析树。

(2) **查询优化器**（CPU 密集）：基于各种规则和统计数据对查询进行评估，尝试构建执行计划。

(3) **查询执行器**（I/O 密集）：遍历数据库存储并找到要返回的元组。

(4) **结果集**（网络密集）：被序列化并被推送至客户端。

显然，所有数据库都需要大量的资源去执行查询。通常，大多数时间实际都花费在查询执行上。而且根据设计，磁盘（不管是旋转磁盘还是 SSD 磁盘）并不能很好地支持并行执行。因此，数据库系统在达到饱和之前，能执行的并发查询的数量是有限制的。这个限制在很大程度上取决于实际使用的数据库引擎以及运行的硬件，还有其他一些不太明显的方面，比如锁、上下文切换以及 CPU 缓存耗尽。我们预期的结果应该是每秒数百个查询，非阻塞 API 能够轻松维持数十万个处于打开状态的 HTTP 连接，相比之下，数据库的处理能力就显得太低了。

我们已经知道数据库的吞吐量严重受限于硬件，采用完全的反应式驱动也没有太大的意义。从技术上讲，你可以基于 Netty 或 RxNetty 实现一个线路协议，避免阻塞客户端线程。实际上，许多非标准的、独立开发的方式（参见 postgresql-async、postgres-async-driver、adbcj 和 finagle–mysql）都尝试用非阻塞网络栈实现特定数据库的线路协议。但是，JVM 只能轻松地处理几百到几千个线程（参见 A.2 节），从头重写已经很完善的 JDBC API 并不会带来太大的益处。Lightbend 常用的反应式技术栈由 Akka 工具包支持，但是它的 Slick 在底层使用的也是 JDBC。另外，还有社区主导的项目致力于弥合 RxJava 和 JDBC 之间的鸿沟，比如 rxjava-jdbc。

关于如何与关系数据库进行交互，我们建议使用一个专用的、经过仔细调优的线程池，将阻塞式代码隔离在这里。这样应用程序的其他部分就能实现高度的反应性，并且只需对少量的线程进行操作。但是从实用的角度来看，还是应当继续使用 JDBC，将其替换为更加反应式的方式会带来很多麻烦，并且没有明显的收益。4.1 节已经给出了在经典软件栈中如何与 JDBC 进行交互的一些提示。即便是基于阻塞式的 JDBC，我们依然可以使用 RxJava 进行一些实验。

在PostgreSQL中使用NOTIFY和LISTEN的案例

PostgreSQL 内置了一个特殊的消息机制，它是通过扩展的 SQL 语句 LISTEN 和 NOTIFY 实现的。每个 PostgreSQL 客户端都可以通过 SQL 语句向虚拟**通道**（channel）发送通知，如下所示。

```
NOTIFY my_channel;
NOTIFY my_channel, '{"answer": 42}';
```

本例首先发送了一条空的通知，随后发送了一个任意的字符串（它可以是 JSON、XML 或者是其他编码格式的数据）到 my_channel 通道。换言之，通道就是一个由 PostgreSQL 数据库引擎管理的队列。比较有意思的是，发送通知是事务的一部分，所以消息的投递会在提交之后进行，如果出现回滚，消息将会被丢弃。

为了消费特定通道的通知，首先要通过 LISTEN 监听该通道。监听指定的连接时，获取通知的唯一办法就是使用 getNotifications() 方法进行定期轮询。这会导致随机的延迟、不必要的 CPU 负载和上下文切换。但令人遗憾的是，它的 API 就是这样设计的。完整的阻塞式代码样例如下所示。

```
try (Connection connection =
    DriverManager.getConnection("jdbc:postgresql:db")) {
        try (Statement statement = connection.createStatement()) {
            statement.execute("LISTEN my_channel");
        }
        Jdbc4Connection pgConn = (Jdbc4Connection) connection;
        pollForNotifications(pgConn);
    }
}

//...

void pollForNotifications(Jdbc4Connection pgConn) throws Exception {
    while (!Thread.currentThread().isInterrupted()) {
        final PGNotification[] notifications = pgConn.getNotifications();
        if (notifications != null) {
            for (final PGNotification notification : notifications) {
                System.out.println(
                        notification.getName() + ": " +
                        notification.getParameter());
            }
        }
        TimeUnit.MILLISECONDS.sleep(100);
    }
}
```

样例不仅会阻塞客户端线程，还必须保持一个 JDBC 连接处于打开状态，因为监听是关联到特定连接的。不过，我们至少可以同时监听多个通道了。上述代码很烦琐，但是很简单直接。在调用 LISTEN 之后，样例会进入一个不会退出的循环，以便于获取新的通知。调用 getNotifications() 是有破坏性的，这意味着它会丢弃返回的通知，所以调用它两次不会返回相同的事件。getName() 能够得到通道的名称（例如 my_channel），getParameter() 则会返回可选的事件内容，比如 JSON 载荷。

这个 API 已经过时了，比如它使用 null 表示没有待处理的通知，使用数组而不是集合。接下来，让它变得对 Rx 更加友好一些。由于缺少推送通知的机制，必须使用非阻塞的 interval() 操作符重新实现轮询。有很多小细节能够让自定义的 Observable 行为更加合理，在以下样例（尚未完成）之后，我们会进一步讨论。

```
Observable<PGNotification> observe(String channel, long pollingPeriod) {
    return Observable.<PGNotification>create(subscriber -> {
        try {
            Connection connection = DriverManager
                    .getConnection("jdbc:postgresql:db");
            subscriber.add(Subscriptions.create(() ->
                    closeQuietly(connection)));
            listenOn(connection, channel);
            Jdbc4Connection pgConn = (Jdbc4Connection) connection;
            pollForNotifications(pollingPeriod, pgConn)
                    .subscribe(Subscribers.wrap(subscriber);
        } catch (Exception e) {
            subscriber.onError(e);
        }
    }).share();
}

void listenOn(Connection connection, String channel) throws SQLException {
    try (Statement statement = connection.createStatement()) {
        statement.execute("LISTEN " + channel);
    }
}

void closeQuietly(Connection connection) {
    try {
        connection.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

如果没有 SQLException，代码将会惊人的短。但是，不要介意，我们的目标是生成健壮的 Observable<PGNotification>。首先，样例推迟了打开数据库连接的行为，真正有人订阅的时候才会执行。同时，为了避免连接泄漏（对于直接处理 JDBC 的应用程序来说，这是非常严重的问题），Subscriber 取消订阅的时候，样例应该确保连接能够正常关闭。另外，流出现错误的时候会取消订阅，从而也能实现连接的关闭。

现在，样例做好了调用 listenOn() 的准备，开始通过打开的连接接收通知。如果在执行这个语句的时候抛出异常，将通过调用 subscriber.onError(e) 捕获和处理异常。它不仅会将错误无缝地传递给订阅者，还会强制关闭连接。但是，如果 LISTEN 请求成功，下一次调用 getNotifications() 将会返回此后发送的所有事件。

我们不想阻塞任何的线程，所以在 pollForNotifications() 中，使用 interval() 创建了一个内部的 Observable。样例使用相同的 Subscriber 订阅该 Observable，但是使用 Subscribers.wrap() 进行了包装，这样 onStart() 就不会在这个 Subscriber 上执行两次。

```java
Observable<PGNotification> pollForNotifications(
            long pollingPeriod,
            AbstractJdbc2Connection pgConn) {
    return Observable
            .interval(0, pollingPeriod, TimeUnit.MILLISECONDS)
            .flatMap(x -> tryGetNotification(pgConn))
            .filter(arr -> arr != null)
            .flatMapIterable(Arrays::asList);
}

Observable<PGNotification[]> tryGetNotification(
            AbstractJdbc2Connection pgConn) {
    try {
        return Observable.just(pgConn.getNotifications());
    } catch (SQLException e) {
        return Observable.error(e);
    }
}
```

我们会定期检查 `getNotifications()` 的内容。首先把它们包装在一个看上去有些笨重的 `Observable<PGNotification[]>` 中。返回的数组 `PGNotification[]` 可能会是 `null`，所以用 `filter()` 过滤掉 `null` 值，然后通过 `flatMapIterable()` 打开数组，先是使用 `Arrays::asList` 将其转换为 `List<PGNotification>`。建议你仔细看一下这些步骤，跟踪中间 `Observable` 的类型。这里之所以会包含 `closeQuietly()` 和 `tryGetNotification()`，是为了处理检查型异常（`SQLException`）。注意，这个异常在 `closeQuietly()` 中以静默的方式处理，因为在该调用的上下文中无法采取其他的措施。例如，如果有人恰好在这时取消订阅，就不能转发该异常。

实现的最后一个细节是 `publish()` 和 `refCount()`，接近第一个方法的结尾。这两个方法能够在多个订阅者之间共享一个 JDBC 连接。如果没有它们，每个新的订阅者都会创建一个新连接并进行监听，这是非常浪费的。另外，`refCount()` 会跟踪订阅者的数量，最后一个订阅者取消订阅的时候，它会关闭数据库连接。参见 2.7.1 节，了解 `publish()` 和 `refCount()` 的更多细节，尤其要关注它如何改变了 `Observable.create()` 中 lambda 表达式的行为。

需要注意，一个连接可以监听多个通道。作为练习，你可以尝试实现 `observe()`，以便在所有订阅者和他们感兴趣的所有通道之间重用相同的连接。如果执行一次 `observe()` 并多次进行订阅，当前的实现能够共享同一个连接。其实它可以一直重用同一个连接，即便订阅者感兴趣的是不同的通道。

其实，探索 PostgreSQL 中 `LISTEN` 和 `NOTIFY` 的价值并不大，因为目前有更快、更健壮、更可靠的消息队列。但是这个案例展现了如何在更加反应式的场景中发挥 JDBC，即便还需要一些阻塞和轮询的机制。

5.4 CompletableFuture与Stream

除了 lambda 表达式、新的 `java.time` API 以及几个小的功能增强，Java 8 还提供了 `CompletableFuture` 类。这个类显著增强了自 Java 5 以来的 `Future`。单纯的 `Future` 代表一个在后台执行的异步操作，通常会由 `ExecutorService` 生成。但是，`Future` 的 API 过于简

单，迫使开发人员一直阻塞对 Future.get() 的调用。我们必须采用一直繁忙等待的方式，才能得到第一个完成的 Future。在这方面，根本没有其他的组合方案。5.4.1 节会简要介绍一下 CompletableFuture 是如何运行的。随后，我们会在 CompletableFuture 和 Observable 之间实现一个很薄的中间层。

5.4.1 CompletableFuture概述

Observable 提供了几十种有用的方法，成功弥合了这一鸿沟，这些方法大多数都是非阻塞和可组合的。我们已经习惯于使用 map() 在运行时异步转换传入的事件。除此之外，Observable.flatMap() 能够将单个事件替换成 Observable，从而将异步任务链接起来。类似的操作可以通过 CompletableFutures 实现。假设有个服务需要两条不相关的信息：User 和 GeoLocation。在获取到它们之后，样例会请求几家独立的旅行社查询航班信息（实体为 Flight），并在返回最快的供应商那里订票（实体为 Ticket）。最后这个需求其实最难实现，在 Java 8 之前，使用 ExecutorCompletionService 才能找到响应最快的任务。

```
User findById(long id) {
    //...
}

GeoLocation locate() {
    //...
}

Ticket book(Flight flight) {
    //...
}

interface TravelAgency {
    Flight search(User user, GeoLocation location);
}
```

使用方式如下。

```
ExecutorService pool = Executors.newFixedThreadPool(10);
List<TravelAgency> agencies = //...

User user = findById(id);
GeoLocation location = locate();
ExecutorCompletionService<Flight> ecs =
    new ExecutorCompletionService<>(pool);
agencies.forEach(agency ->
    ecs.submit(() ->
        agency.search(user, location)));
Future<Flight> firstFlight = ecs.poll(5, SECONDS);
Flight flight = firstFlight.get();
book(flight);
```

在 Java 开发人员中，ExecutorCompletionService 并不流行，而且有了 CompletableFuture 之后，根本就不再需要它了。但是，在这里首先会看到 ExecutorCompletionService 怎样包装了 ExecutorService。包装完成之后，就可以通过 poll 方法在完成的任务到达时获取结

果。如果单纯地使用 ExecutorService，会得到一堆 Future 对象，完全不知道哪一个会率先完成，所以 ExecutorCompletionService 的作用就发挥了出来。但是，这里依然需要一个额外的线程来阻塞等待 TravelAgency 的响应。另外，我们并没有充分发挥并发的优势（同时加载 User 和 GeoLocation）。

重构会首先将所有的方法转换成异步方法，然后对 CompletableFuture 进行相应的组合。通过这种方式，代码将会变成完全非阻塞的（主线程几乎可以立即完成），可以尽可能地实现并行化。

```
CompletableFuture<User> findByIdAsync(long id) {
    return CompletableFuture.supplyAsync(() -> findById(id));
}

CompletableFuture<GeoLocation> locateAsync() {
    return CompletableFuture.supplyAsync(this::locate);
}

CompletableFuture<Ticket> bookAsync(Flight flight) {
    return CompletableFuture.supplyAsync(() -> book(flight));
}

@Override
public CompletableFuture<Flight> searchAsync(User user, GeoLocation location) {
    return CompletableFuture.supplyAsync(() -> search(user, location));
}
```

以上只是使用异步 CompletableFuture 将阻塞式代码包装了起来。supplyAsync() 方法会接收一个可选的 Executor 作为参数。如果没有指定，那么将会使用 ForkJoinPool.commonPool() 中定义的全局 Executor。建议始终使用自定义的 Executor，但样例这里就使用默认的 Executor 了。需要注意，这个默认的 Executor 会在所有的 CompletableFuture、并行流（参见 8.5 节）和其他几个不太明显的位置共享。

```
import static java.util.function.Function.identity;


List<TravelAgency> agencies = //...
CompletableFuture<User> user = findByIdAsync(id);
CompletableFuture<GeoLocation> location = locateAsync();

CompletableFuture<Ticket> ticketFuture = user
    .thenCombine(location, (User us, GeoLocation loc) -> agencies
        .stream()
        .map(agency -> agency.searchAsync(us, loc))
        .reduce((f1, f2) ->
            f1.applyToEither(f2, identity())
        )
        .get()
    )
    .thenCompose(identity())
    .thenCompose(this::bookAsync);
```

在上面的代码片段中发生了很多的事情。完整地阐述 CompletableFuture 已经超出了本

书的范围，但是有些 API 在 RxJava 环境中是非常有用的。首先，样例异步地获取 User 和 GeoLocation 信息。这两个操作是独立的，可以并发运行。但是想要获取它们的结果，显然不能阻塞和浪费客户端线程。这就是 thenCombine() 做的事情。它接收两个 CompletableFuture（user 和 location），并在两者都完成的时候，异步执行一个回调。有意思的是，这个回调可以返回一个值，这个值将会成为最终形成的 CompletableFuture 的新内容，如下所示。

```
CompletableFuture<Long> timeFuture = //...
CompletableFuture<ZoneId> zoneFuture = //...

CompletableFuture<Instant> instantFuture = timeFuture
        .thenApply(time -> Instant.ofEpochMilli(time));

CompletableFuture<ZonedDateTime> zdtFuture = instantFuture
        .thenCombine(zoneFuture, (instant, zoneId) ->
                ZonedDateTime.ofInstant(instant, zoneId));
```

CompletableFuture 与 Observable 有很多相似之处。thenApply() 会对 Future 返回的值执行动态转换，这类似于 Observable.map()。样例提供了一个从 Long 到 Instant（Instant::ofEpochMilli）的函数，从而将 CompletableFuture<Long> 转换为 CompletableFuture<Instant>。随后，接收两个 Future（instantFuture 和 zoneFuture），使用 thenCombine() 对它们的值（Instant 和 ZoneId）进行转换。这次转换会返回 ZoneDateTime，但是因为大多数的 CompletableFuture 操作符都是非阻塞的，在返回值中得到的是 CompletableFuture<ZonedDateTime>，这个过程与 Observable 中的 zip() 非常类似。我们回到之前订票的样例，下面的代码片段可能会有些晦涩难懂。

```
List<TravelAgency> agencies = //...

agencies
    .stream()
    .map(agency -> agency.searchAsync(us, loc))
    .reduce((f1, f2) ->
            f1.applyToEither(f2, identity())
    )
    .get()
```

样例需要在每个 TravelAgency 上通过调用 searchAsync() 来启动异步操作。调用之后，马上就会得到一个 List<CompletableFuture<Flight>>。如果只想要第一个完成的 Future，这个数据结构操作起来就非常不便利了。为了解决这个问题，样例可以使用 CompletableFuture.allOf() 和 CompletableFuture.anyOf() 这样的方法。在语义上，后者恰好是需要的。它会接收一组 CompletableFuture，并在第一个底层的 CompletableFuture 完成之后，返回一个 CompletableFuture，然后丢弃所有其他的 CompletableFuture。这类似于 Observable.amb()（参见 3.2.3 节）。令人遗憾的是，anyOf() 的语法有些诡异。首先，它会接收一个数组（可变参数），不管底层 Future 是什么类型（比如 Flight），它始终会返回 CompletableFuture<Object>。我们可以使用它，但是代码会变得相当杂乱，如下所示。

```
.thenCombine(location, (User us, GeoLocation loc) -> {
    List<CompletableFuture<Flight>> fs = agencies
        .stream()
```

```
            .map(agency -> agency.searchAsync(us, loc))
            .collect(toList());
        CompletableFuture[] fsArr = new CompletableFuture[fs.size()];
        fs.toArray(futuresArr);
        return CompletableFuture
            .anyOf(futuresArr)
            .thenApply(x -> ((Flight) x));
    })
```

接下来介绍一下 `Stream.reduce()` 的技巧。`CompletableFuture.applyToEither()` 操作符接收两个 `CompletableFuture`，并将给定的转换函数应用到第一个完成的 `CompletableFuture` 上。如果你有两个同种类型的任务，并且只关心第一个完成的，那么 `applyToEither()` 转换是非常有用的。在下面的样例中，我们在两个服务器上查询 `User`：主服务器和备用服务器。不管哪个率先完成，样例都会使用一个简单的转换，将用户的出生日期抽取出来。第二个 `CompletableFuture` 的执行不会中断，但是结果会被忽略。显然，最后得到的是 `CompletableFuture<LocalDate>`。

```
CompletableFuture<User> primaryFuture = //...
CompletableFuture<User> secondaryFuture = //...

CompletableFuture<LocalDate> ageFuture =
    primaryFuture
        .applyToEither(secondaryFuture,
            user -> user.getBirth());
```

`applyToEither()` 只能与两个 `CompletableFuture` 组合使用，而看上去有点怪异的 `anyOf()` 能够接收任意数量的 `CompletableFuture`。幸而，还可以使用前两个 `Future` 来调用 `applyToEither()`，然后将得到的结果（前两个中较快的一个）应用到第三个上游 `Future` 中（从而得到前三个中最快的一个）。通过递归调用 `applyToEither()`，我们能够得到整体最快的 `CompletableFuture`。这个便利的技巧可以通过 `reduce()` 操作符来实现。最后一个提示是 `Function` 中的 `identity()` 方法，这是 `applyToEither()` 需要的，我们必须提供一个转换功能来处理得到的第一个结果。如果想要**原样**保留结果，那么可以使用一个恒等函数，写成 `f -> f` 或 `(Flight f) -> f` 的形式。

最终实现的 `CompletableFuture<Flight>` 会在最快的 `TravelAgency` 响应时完成，这个过程是异步进行的。`thenCombine()` 的结果还有点小问题。不管传递给 `thenCombine()` 的转换内容是什么，返回的结果都会包装在一个 `CompletableFuture` 中。我们的场景中返回的是 `CompletableFuture<Flight>`，所以 `thenCombine()` 的结果类型就是 `CompletableFuture<CompletableFuture<Flight>>`。在使用 `Observable` 的时候，双重包装也是很常见的问题，可以使用相同的技巧来应对这两种情况：`flatMap()`!（参见 3.1.2 节）。但是，就像 `map()` 在 `Future` 中被称为 `thenApply()` 一样，`flatMap()` 也被称为 `thenCompose()`。

```
Observable<Observable<String>> badStream = //...
Observable<String> goodStream = badStream.flatMap(x -> x);

CompletableFuture<CompletableFuture<String>> badFuture = //...
CompletableFuture<String> goodFuture = badFuture.thenCompose(x -> x);
```

我们通常使用 `flatMap()/thenCompose()` 来链接异步计算，但是这里只是简单地用来拆解不

正确的类型。需要注意，thenCompose() 运行给定的转换会返回 CompletableFuture 类型。但是，因为内部类型已经是 Future 了，样例可以使用 identity() 或简单的 x -> x，拆解内部的 Future 以修正类型的问题。

最后得到了 CompletableFuture<Flight>（缩写为 flightFuture）。接下来，样例就可以调用 bookAsync() 了，它以 Flight 作为参数。

```
CompletableFuture<Ticket> ticketFuture = flightFuture
        .thenCompose(flight -> bookAsync(flight));
```

样例在调用 bookAsync() 时，使用 thenCompose() 就更加自然了。这个方法返回的是 CompletableFuture<Ticket>，为了避免双重包装，我们选择使用 thenCompose()，而不是 thenApply()。

5.4.2　与CompletableFuture进行交互

返回 Observable<T> 的工厂方法 Observable.from(Future<T>) 已经存在了。但是，由于旧有 Future<T> API 的限制，它有一些缺点，其中最大的缺点就是 Future.get() 内部是阻塞的。传统的 Future<T> 实现没有办法注册回调并以异步的方式进行处理，因此在反应式应用程序中，它们并没有太大的用处。

CompletableFuture 则与之完全不同。在语义上讲，可以将 CompletableFuture 视为具有如下特征的 Observable。

- **它是 hot 类型的。**
  CompletableFuture 的后台计算是立即启动的，无论是否有人注册 thenApply() 这样的回调。
- **它的结果是缓存的。**
  CompletableFuture 的后台计算会立即触发，得到的结果会被转发至所有注册的回调。除此之外，如果有的回调是在 CompletableFuture 完成之后注册的，这个回调会立即基于完成时的值（或异常）进行调用。
- **它只能发布一个值或异常。**
  理论上，Future<T> 只能完成一次（或者永远不进入完成状态），并且带有 T 类型的返回值或异常。这符合 Observable 的契约。

1. 将CompletableFuture转换成只有一个元素的Observable

首先编写一个工具函数，这个函数接收 CompletableFuture<T> 并返回一个正常的 Observable<T>。

```
class Util {
    static <T> Observable<T> observe(CompletableFuture<T> future) {
        return Observable.create(subscriber -> {
            future.whenComplete((value, exception) -> {
                if (exception != null) {
                    subscriber.onError(exception);
                } else {
                    subscriber.onNext(value);
                    subscriber.onCompleted();
```

```
                    }
                });
            });
        }
    }
```

为了同时能够得到成功和失败的完成通知，这里使用了 CompletableFuture.whenComplete() 方法，它接收两个相互排斥的参数。如果 exception 不为空，就代表底层的 Future 失败了；否则，样例就能得到成功的 value 值。这两种情况都会通知传入的 subscriber。需要注意，如果订阅是（以某种方式）在 CompletableFuture 完成之后才出现的，那么回调将会立即执行。CompletableFuture 会缓存完成时的结果，所以在此之后注册的回调会立即在客户端线程中执行。

很容易会想到注册一个取消订阅处理器，在取消订阅的同时，取消对 CompletableFuture 的执行。

```
//不要这样做
subscriber.add(Subscriptions.create(
        () -> future.cancel(true)));
```

这是一个很糟糕的主意。基于一个 CompletableFuture 可以创建很多 Observable，而每个 Observable 可能又会有多个 Subscriber。如果有一个 Subscriber 在 Future 完成之前取消执行，将会影响到其他所有的 Subscriber。

需要记住，按照 Rx 的语义，CompletableFuture 是 hot 类型的，并且会进行**缓存**。CompletableFuture 会立即进行计算，而 Observable 只在有人订阅的时候才会开始计算。下面的小工具可以进一步优化 API。

```
Observable<User> rxFindById(long id) {
    return Util.observe(findByIdAsync(id));
}

Observable<GeoLocation> rxLocate() {
    return Util.observe(locateAsync());
}

Observable<Ticket> rxBook(Flight flight) {
    return Util.observe(bookAsync(flight));
}
```

显然，如果你消费的 API 从一开始就支持 Observable，那么就没有必要使用这些额外的适配器层了。但是，如果具备的只是 CompletableFuture，将它们转换成 Observable 才是高效和安全的。RxJava 的优势在于它能够以更加简洁的方式解决最初的问题。

```
Observable<TravelAgency> agencies = agencies();
Observable<User> user = rxFindById(id);
Observable<GeoLocation> location = rxLocate();

Observable<Ticket> ticket = user
    .zipWith(location, (us, loc) ->
        agencies
            .flatMap(agency -> agency.rxSearch(us, loc))
```

```
                .first()
        )
        .flatMap(x -> x)
        .flatMap(this::rxBook);
```

使用 RxJava API 的客户端代码看上去更简洁，也更易读。Rx 原生地支持流形式的“带有多个值的 future”。如果你还是觉得 flatMap() 中的恒等转换 x -> x 有些烦琐，可以结合 Pair 辅助容器对 zipWith() 进行切分，如下所示。

```
import org.apache.commons.lang3.tuple.Pair;

//...
Observable<Ticket> ticket = user
        .zipWith(location, (usr, loc) -> Pair.of(usr, loc))
        .flatMap(pair -> agencies
                .flatMap(agency -> {
                    User usr = pair.getLeft();
                    GeoLocation loc = pair.getRight();
                    return agency.rxSearch(usr, loc);
                }))
        .first()
        .flatMap(this::rxBook);
```

此时，你应该能够理解为何不需要额外的 x -> x。zipWith() 会接收两个独立的 Observable，并异步等待它们。Java 没有内置的配对和元组类，所以这里必须要提供一个转换函数，用来接收两个流中的事件并将它们组合成一个 Observable<Pair<User, Location>> 对象。这个对象将会作为下游 Observable 的输入。随后，根据给定的 User 和 Location，使用 flatMap() 并发查询每个旅行社。flatMap() 会拆解结果（从语法角度来说），所以最终形成的流就是一个简单的 Observable<Flight>。这两种情况都会调用 first()，但是样例只处理上游流中第一个出现的 Flight（最快的 TravelAgency）。

2. 从Observable到CompletableFuture

在有些情况下，你使用的 API 可能支持 CompletableFuture，但是不支持 RxJava。这很常见，尤其是考虑到前者是 JDK 的一部分，而后者只是一个库。在这样的情况下，如果能够将 Observable 转换成 CompletableFuture，那是非常棒的，以下两种方式能够实现这样的转换。

- Observable<T> 到 CompletableFuture<T>

  预期在流中发布一个条目时使用它，比如 Rx 包装一个方法调用或者请求 / 响应模式。如果流完成的时候只发布了一个值，那么 CompletableFuture<T> 也会成功完成。显然，如果流出现异常或者完成时发布的值不是一个，那么 future 将会以异常的状态完成。

- Observable<T> 到 CompletableFuture<List<T>>

  在这种场景下，上游 Observable 的值都已发布并且流完成的时候，CompletableFuture 才会完成。这只是上一个转换的一种特殊形式。

使用下面的工具方法，可以很容易地实现第一个场景。

```
static <T> CompletableFuture<T> toFuture(Observable<T> observable) {
    CompletableFuture<T> promise = new CompletableFuture<>();
```

```
        observable
                .single()
                .subscribe(
                        promise::complete,
                        promise::completeExceptionally
                );
        return promise;
    }
```

在深入讲解实现细节之前，需要注意这个转换有一个很重要的副作用：它会订阅 Observable，因此会强制 cold 类型的 Observable 进行评估和计算。另外，对这个转换的每次调用都会再次进行订阅，这只是你需要注意的一个设计选择。

除此之外，功能实现本身是非常有意思的。首先，使用 single() 强制 Observable 只发布一个元素，否则将会抛出异常。如果这个流在发布完一个值之后就完成，我们就调用 CompletableFuture.complete() 方法。事实证明，可以从头创建 CompletableFuture，而不需要任何的后台线程和异步任务。它依然是一个 CompletableFuture，但是实现完成并通知所有已注册回调的唯一方法是显式调用 complete()。这是一种异步交换数据的有效方法，至少在 RxJava 不可用的时候是这样的。

如果出现失败，我们可以通过调用 CompletableFuture.completeExceptionally() 触发所有已注册回调的错误处理功能。你可能觉得有点意外，这就是完整的实现。toFuture 返回的 Future 看上去似乎有与其关联的后台任务，但实际上我们会显式地完成它。

从 Observable<T> 到 CompletableFuture<List<T>> 的转换非常简单。

```
    static <T> CompletableFuture<List<T>> toFutureList(Observable<T> observable) {
        return toFuture(observable.toList());
    }
```

CompletableFuture 和 Observable 的交互是非常有用的。前者的设计非常好，但是缺乏后者的表述性和功能丰富性。因此，如果必须要在基于 RxJava 中的应用程序中处理 CompletableFuture，那么可以尽早使用这些简单的转换功能，以提供一致和可预测的 API。请确保你理解了 Future 立即执行（hot）和 Observable 默认延迟执行的区别。

5.5 Observable与Single

经常看到有人害怕使用 RxJava，原因在于它看上去过于以流为中心。Observable 是一个流，甚至可能是无穷流，而且所有的操作符都用流的术语进行描述。List<T> 可能只包含一个元素，与之类似，有些 Observable<T> 按照定义只能发布一个事件。如果 List<T> 始终只包含一个元素，会让人感觉非常困惑，所以这种场景下可以简单地使用 T 或 Optional<T>。在 RxJava 领域，有一个针对仅发布一个事件的 Observable 抽象，它就是 rx.Single<T>。

Single<T> 基本上是一个容器，包含了一个未来会出现的 T 类型的值或 Exception。就这方面，Java 8 中的 CompletableFuture 可以说是 Single 的近亲（参见 5.4 节）。但是，与 CompletableFuture 不同，Single 是延迟执行的，只有有人实际订阅的时候才会生成值。Single 主要用于异步返回单个值（duh!）并具有高失败概率的 API。显然，在涉及 I/O 通

信的请求 – 响应类型场景中（如网络调用），Single 是一个很好的替代方案。相对于正常的网络调用，延迟一般会非常高，失败也是难以避免的。另外，因为 Single 是延迟执行并且是异步的，可以使用各种技巧来改善延迟和弹性，比如并发调用独立的操作并将结果组合在一起（参见 5.5.2 节）。Single 通过类型级别的提示，减轻了 API 返回 Observable 带来的困惑。

```
Observable<Float> temperature() {
    //...
}
```

我们很难预测上面方法的契约。只返回一个温度值就完成吗？还是返回温度值形成的一个无穷流？甚至更糟糕，在某些场景下，它可能不发布任何事件就完成了。如果 temperature() 返回 Single<Float>，我们马上就能知道预期的输出是什么。

5.5.1 创建和消费Single

在支持的操作符方面，Single 与 Observable 非常相似，所以我们不会在这方面花费太多时间。而是将 Single 的操作符和 Observable 对应的操作符进行简单对比，并且主要关注 Single 的用例。创建 Single 的方式很少，让我们从常量形式的 just() 和 error() 操作符开始。

```
import rx.Single;


Single<String> single = Single.just("Hello, world!");
single.subscribe(System.out::println);

Single<Instant> error =
        Single.error(new RuntimeException("Opps!"));
error
        .observeOn(Schedulers.io())
        .subscribe(
                System.out::println,
                Throwable::printStackTrace
        );
```

just() 操作符没有接收多个值的重载版本，毕竟按照定义 Single 只能持有一个值。同时，subscribe() 方法会接收两个参数，而不是三个，因为它没有必要再有 onComplete() 回调了。Single 在完成的时候要么会有一个值（第一个回调），要么会出现异常（第二个回调）。仅仅监听完成通知等价于订阅单个值。除此之外，这里还包含了 observeOn() 操作符，它的运行方式和 Observable 对应的操作符是完全相同的。同样的情况也适用于 subscribeOn()（参见 4.5 节）。最后，你可以使用 error() 操作符来创建 Single，它始终会以给定 Exception 的形式完成。

接下来实现一个更加现实的场景，也就是发送 HTTP 请求。5.2 节介绍了如何使用 RxNetty 构建异步的 HTTP 客户端。这次我们要使用 async-http-client，它也在底层使用了 Netty。发送 HTTP 请求之后，样例可以提供一个回调实现，这个回调会在请求得到响应或出现错误时异步调用。这种场景非常符合创建 Single。

```java
import com.ning.http.client.AsyncCompletionHandler;
import com.ning.http.client.AsyncHttpClient;
import com.ning.http.client.Response;


AsyncHttpClient asyncHttpClient = new AsyncHttpClient();

Single<Response> fetch(String address) {
    return Single.create(subscriber ->
            asyncHttpClient
                    .prepareGet(address)
                    .execute(handler(subscriber)));
}

AsyncCompletionHandler handler(SingleSubscriber<? super Response> subscriber) {
    return new AsyncCompletionHandler() {
        public Response onCompleted(Response response) {
            subscriber.onSuccess(response);
            return response;
        }

        public void onThrowable(Throwable t) {
            subscriber.onError(t);
        }
    };
}
```

Single.create() 与 Observable.create()（参见 2.4.1 节）非常类似，但是它有一个非常重要的限制：调用 onSuccess() 或者 onError() 的机会只有一次。从技术上，样例可以创建一个永不完成的 Single，但是不允许多次调用 onSuccess()。除了 Single.create() 之外，你还可以尝试一下 Single.fromCallable()，后者能够接收一个 Callable<T> 并返回 Single<T>。这同样非常简单。

回到 HTTP 客户端的样例。响应返回的时候，样例会通过调用 onSuccess() 告知订阅者，或者在异步失败回调中通过 onError() 传递异常。可以采用与 Observable 类似的方式来使用 Single。如下所示。

```java
Single<String> example =
    fetch("http://www.example.com")
        .flatMap(this::body);

String b = example.toBlocking().value();

//...

Single<String> body(Response response) {
    return Single.create(subscriber -> {
        try {
            subscriber.onSuccess(response.getResponseBody());
        } catch (IOException e) {
            subscriber.onError(e);
        }
    });
```

```
}

//与body()的功能相同
Single<String> body2(Response response) {
    return Single.fromCallable(() ->
        response.getResponseBody());
}
```

比较遗憾的是，Response.getResponseBody() 可能会抛出 IOException，所以不能简单地使用 map(Response::getResponseBody)。但至少我们在这里看到了 Single.flatMap() 是如何运行的。将有潜在危险的 getResponseBody() 方法用 Single<String> 包装之后，就能确保潜在的失败会封装起来，并在类型系统中清晰地进行表述。在了解 Observable.flatMap() 之后，Single.flatMap() 的运行方式和预期是一样的：如果第二个阶段的计算（在本例中是 this::body）失败，整个 Single 也会失败。有意思的是，Single 有 map() 和 flatMap() 操作符，但是没有 filter() 操作符。你能猜到为什么吗？原理上，filter() 可能过滤掉 Single<T> 中不满足给定 Predicate<T> 的内容。Single<T> 内部**必须**有且仅有一个条目，而 filter() 有可能会导致 Single 内部一无所有。

与 BlockingObservable（参见 4.2 节）类似，Single 有自己的 BlockingSingle，可以通过 Single.toBlocking() 来创建。类似地，创建 BlockingSingle<T> 并不会阻塞。但是样例调用 value() 会阻塞，直到类型为 T（在本例中是包含响应的 String）的值可用。如果出现异常，它会从 value() 方法中重新抛出。

5.5.2 使用zip、merge和concat组合响应

如果没有可组合的操作符，那么 rx.Single 不会有太大的用处。你遇到的最重要的操作符可能就是 Single.zip()，它的运行方式和 Observable.zip() 完全相同（参见 3.2.1 节），但是语义更简单。Single 始终只会发布一个值（或异常），所以 Single.zip()（或实例版本的 Single.zipWith() 方法）的结果只会有一个配对 / 元组。zip() 基本上就是底层两个 Single 都完成的时候，创建第三个 Single 的一种方法。[2]

假设要在网站上渲染一篇文章。满足这个请求需要三个独立的操作：从数据库读取文章内容；请求社交媒体网站，收集到目前为止的点赞人数；更新文章阅读数量指标。原生实现不仅会序列化地执行这三个操作，如果某个步骤非常慢，还有可能导致无法接受的延迟。借助 Single，可以对每个步骤分别建模，如下所示。

```
import org.springframework.jdbc.core.JdbcTemplate;

//...

Single<String> content(int id) {
    return Single.fromCallable(() -> jdbcTemplate
        .queryForObject(
            "SELECT content FROM articles WHERE id = ?",
            String.class, id))
```

注 2：在 CompletableFututre 中，这个操作符为 thenCombine。

```
        .subscribeOn(Schedulers.io());
}

Single<Integer> likes(int id) {
    //对社交媒体Web站点的异步HTTP请求
}

Single<Void> updateReadCount() {
    //Single只有副作用，没有返回值
}
```

这个样例展示了如何使用 `fromCallable` 创建 `Single`，其中，`fromCallable` 通过 lambda 表达式传递。这个工具是非常有用的，它管理着错误处理功能（参见 7.1.1 节）。`content()` 方法使用了 Spring 框架提供的 `JdbcTemplate` 工具，以便从数据库中隐蔽地加载文章内容。JDBC 本质上会阻塞 API，所以样例显式调用了 `subscribeOn()`，让 `Single` 变成异步的。样例省略了 `likes()` 和 `updateReadCount()` 的实现。你可以将 `likes()` 设想为通过 RxNetty（参见 5.2 节）向特定的 API 发送异步 HTTP 请求。`updateReadCount()` 比较有意思，它的类型是 `Single<Void>`。这表明它会执行一些没有返回值的副作用，但是这个操作会有明显的延迟。不过，我们依然想要在异步发生可能的失败时得到通知。针对这种情况，RxJava 也有一个专门的类型：`Completable`。它会列明是正常完成而没有结果，还是异步地产生了异常。

将这三个操作通过 `zip` 组合起来就非常简单了。

```
Single<Document> doc = Single.zip(
        content(123),
        likes(123),
        updateReadCount(),
        (con, lks, vod) -> buildHtml(con, lks)
);

//...

Document buildHtml(String content, int likes) {
    //...
}
```

`Single.zip()` 会接收三个 `Single`（它有接收两个到九个实例的重载版本），三个 `Single` 都完成的时候，就会执行自定义的函数。这个自定义函数的输出会放回到一个 `Single<Document>` 中，样例可以对它进行进一步的转换。你应该注意到 `Void` 结果没有在转换过程中使用，这意味着样例需要等待 `updateReadCount()` 完成，但是并不需要它的结果（为空）。这可能是一种需求，也可能是潜在的优化点：如果 `updateReadCount()` 异步执行，而不等待它正常完成或出现失败，那么构建 HTML 文档的功能依然能够很好地运行。

现在假设一下，如果调用 `likes()` 出现了失败，或者经历了一段长到无法接受的时间才完成（后者会更糟糕），会出现什么情况呢？如果不采用 Reactive Extensions 渲染，HTML 将会完全失败或者经历相当长的时间。但是，我们现在的实现也好不到哪里去。`Single` 支持多个像 `timeout()`、`onErrorReturn()` 和 `onErrorResumeNext()` 这样的操作符，它们能够强化弹性和错误处理的相关功能。这些操作符的行为与 `Observable` 对应的操作符完全一样。

5.5.3 与Observable和CompletableFuture的交互

从类型系统的角度，Observable 和 Single 是互不相关的。这基本意味着，需要 Observable 的时候，我们不能使用 Single，反之亦然。但是在以下两种场景中，它们之间的转换是有意义的。

- 使用 Single 作为 Observable，只发布一个值和完成通知（或错误通知）。
- Single 缺少 Observable 支持的操作符的时候，cache() 就是一个例子。[3]

本节以第二个原因进行举例说明。

```
Single<String> single = Single.create(subscriber -> {
    System.out.println("Subscribing");
    subscriber.onSuccess("42");
});

Single<String> cachedSingle = single
        .toObservable()
        .cache()
        .toSingle();

cachedSingle.subscribe(System.out::println);
cachedSingle.subscribe(System.out::println);
```

这里使用了 cache() 操作符，这样 Single 只会为第一个订阅者生成 42。Single.toObservable() 是一个非常安全和易于理解的操作符。它会接收一个 Single<T> 实例，并将其转换为 Observable<T>，这个 Observable 发布一个元素之后会立即发送完成通知（如果符合 Single 的完成方式则发送错误通知）。与之相反，Observable.toSingle()（不要与 single() 操作符混淆，参见 3.3.3 节）则更加需要注意。与 single() 类似，如果底层 Observable 发布的元素超出了一个，toSingle() 将会抛出一个异常，表明“Observable 发布了太多的元素”；如果 Observable 为空，预期结果将是“Observable 没有发布任何条目”。

```
Single<Integer> emptySingle =
        Observable.<Integer>empty().toSingle();
Single<Integer> doubleSingle =
        Observable.just(1, 2).toSingle();
```

你可能认为 toObservable() 和 toSingle() 使用位置接近的时候，后者会更安全，但事实可能并非如此。比如，中间的 Observable 可能会对 Single 发布的值进行重复或丢弃。

```
Single<Integer> ignored = Single
        .just(1)
        .toObservable()
        .ignoreElements() //有问题的
        .toSingle();
```

在上面的代码中，Observable 中的 ignoreElements() 会丢弃 Single 发布的单个值。因此，使用 toSingle() 操作符的时候，只能看到一个没有任何条目就结束的 Observable。需要注意，toSingle() 和之前看到的其他操作符类似，都是延迟执行的。有人真正订阅的时候，

注 3：截至本文撰写时（RxJava 1.1.6）。这组可用的操作符更新得很快，请确保使用的是最新版本并再次检查。

才会出现 Single 不能精确发布一个事件的异常。

5.5.4　何时使用Single

了解完 Observable 和 Single 这两种抽象，识别它们的差异并理解使用的场景就非常重要了。就像数据结构一样，一种结构不能放之四海皆准。在如下的场景中，应该使用 Single。

- 某种操作必须以某个特定值或异常的形式完成。例如，调用 Web 服务，要么会得到一个来自外部服务器的响应，要么得到某种形式的异常。
- 在你的问题域中，并没有**流**这样的存在，使用 Observable 会造成误导和大材小用。
- Observable 过于重量级，你**衡量**后认为 Single 在某个特定的问题中更快。

另一方面，在如下的场景中优先选择 Observable。

- 你要为某种类型的事件（消息、GUI 事件）进行建模，按照定义，它们会多次出现，甚至是无穷的。
- 或者完全相反，你的预期是：值在完成之前可能出现，也可能不出现。

第二个场景是非常有意思的。你是否会认为，针对某些存储库（repository）中的 findById(int) 方法，使用 Single<Record> 会比使用 Record 或 Observable<Record> 更合理呢？听上去似乎很有道理：我们根据 ID 获取一个条目（这说明这样的 Record 只有一个）。但是，无法确保每个给定的 ID 都一定存在这样一个 Record。因此，这个方法有可能不返回任何内容。我们将其建模为 null、Optional<Record> 或 Observable<Record>，都能很好地处理空流之后紧跟着完成通知的场景。Single 又怎样呢？它要么以一个值的形式（Record）正常完成，要么出现一个异常。如果你想将条目不存在建模为异常，这是你的设计选择，但通常认为这是一种糟糕的实践。判断一条给定 ID 的缺失记录是否是真正的**异常**场景，不应该是存储库层的责任。

5.6　小结

第 2 章和第 3 章让你对 RxJava 有了一个整体的印象和感觉。本章讨论了设计完整反应式应用程序的高级话题。这部分是更高级的内容，介绍了实现事件驱动系统的实际技术，而且不会引入额外的复杂性。本章展现了一些测试基准，证明 RxJava 与非阻塞网络技术栈（如 Netty）组合带来的性能优势。你并非必须要使用这些高级的库，但是如果你想要在商业服务器上实现吞吐量最大化，使用它们肯定会物有所值。

第 6 章

流控制和回压

托马什·努尔凯维茨（Tomasz Nurkiewicz）

到目前为止，我们已经非常熟悉 RxJava 基于推送的特点。事件在上游中的某个地方产生，然后被所有的订阅者消费。我们一直没有关心，如果 `Observer` 的处理比较慢，跟不上 `Observable.create()` 的事件发布节奏，会导致什么后果。本章会就这个问题进行讨论。

RxJava 有以下两种方式来应对生产者比订阅者更活跃的情况。

- 各种流控制机制，比如基于内置的操作符实现采样和批处理。
- 订阅者通过一个名为回压（backpressure）的反馈通道，来传递它们的需求，并且只请求它们能够处理的条目。

本章将会介绍这两种机制。

6.1 流控制

在 RxJava 实现回压（参见 6.2 节）之前，生产者（`Observable`）的生产速度超过消费者（`Observer`）的消费速度一直是困扰我们的问题。RxJava 发明了很多操作符来处理生产者推送事件太多的情况，它们中的大多数本身就非常有趣。有些用于事件的批处理，有些则用于丢弃一些事件。本节将会介绍这些操作符，包括一些样例。

6.1.1 定期采样和节流

在有些场景下，肯定想要接收并处理上游 `Observable` 推送的每个事件。但是，也有一些场景定期采样就足够了。最为典型的场景就是接收某个设备的度量数据，比如温度（对

比 3.3.4 节）。设备生成新度量数据的频率通常与我们无关，尤其是当度量数据频繁出现，而且彼此非常相似的时候。`sample()` 操作符会定期（比如，每秒一次）查看上游的 `Observable`，并将遇到的最新事件发布出来。如果在过去的一秒内没有事件，则不会对下游进行采样。1 秒之后，再继续下一次的采样操作，如下所示。

```
long startTime = System.currentTimeMillis();
Observable
    .interval(7, TimeUnit.MILLISECONDS)
    .timestamp()
    .sample(1, TimeUnit.SECONDS)
    .map(ts -> ts.getTimestampMillis() - startTime + "ms: " + ts.getValue())
    .take(5)
    .subscribe(System.out::println);
```

上述代码将会打印出类似下面的输出。

```
1088ms: 141
2089ms: 284
3090ms: 427
4084ms: 569
5085ms: 712
```

第一列展现了从订阅到采样发布的相对时间。可以清晰地看到，第一次采样发生的时间稍微超出了 1 秒（作为 `sample()` 操作符的要求），而随后的采样基本都是间隔 1 秒。更重要的是打印出的值。`interval()` 操作符从零开始每 7 毫秒发布自然数。因此，在第一次采样的时候，预计会出现 142（1000/7）个事件，而第 142 个值就是 141（从零开始的）。

接下来探讨一个更复杂的采样样例。假设有一个人名的列表，它们会在固定的时间间隔之后分别出现，如下所示。

```
Observable<String> names = Observable
    .just("Mary", "Patricia", "Linda",
        "Barbara",
        "Elizabeth", "Jennifer", "Maria", "Susan",
        "Margaret", "Dorothy");

Observable<Long> absoluteDelayMillis = Observable
    .just(0.1, 0.6, 0.9,
        1.1,
        3.3, 3.4, 3.5, 3.6,
        4.4, 4.8)
    .map(d -> (long)(d * 1_000));

Observable<String> delayedNames = names
    .zipWith(absoluteDelayMillis,
        (n, d) -> Observable
                .just(n)
                .delay(d, MILLISECONDS))
    .flatMap(o -> o);

delayedNames
    .sample(1, SECONDS)
    .subscribe(System.out::println);
```

首先，样例构建了一个人名的序列，然后是一个绝对延迟（以秒为单位，随后映射为毫秒）序列。使用 zipWith() 操作符定义特定人名的延迟，比如 Mary 会在订阅后间隔 100 毫秒出现，而 Dorothy 会在 4.8 秒之后出现。sample() 操作符会阶段性地（每秒）从流中获取上一个时间段内最后出现的人名。所以，第一秒之后，样例通过 println 打印出 Linda，再往后一秒打印出 Barbara。在订阅之后的 2000 到 3000 毫秒之间，没有人名出现，因此 sample() 不发布任何内容。在 Barbara 发布出来两秒后就会看到 Susan。sample() 会转发完成通知（以及错误），并且丢弃最后一个周期。如果我们想让 Dorothy 出现，可以人为地延迟完成通知的发送，如下所示。

```
static <T> Observable<T> delayedCompletion() {
    return Observable.<T>empty().delay(1, SECONDS);
}

//...

delayedNames
    .concatWith(delayedCompletion())
    .sample(1, SECONDS)
    .subscribe(System.out::println);
```

sample() 有一个更高级的变种形式，它接受 Observable 作为参数，而不是一个固定的周期。这里的第二个 Observable（即采样器，sampler）基本上决定了何时从上游源中采样：每次采样器发布任意值，就会采集一个新的样本（前提是自上次采样之后出现了新的值）。你可以使用这个重载版本的 sample()，以动态改变采样率或者只在某个特定的时间点进行采样。例如，在帧重绘或者键盘按下时，截取快照。下面这个样例能够通过 interval() 操作符简单模拟固定周期。

```
//等价的
obs.sample(1, SECONDS);
obs.sample(Observable.interval(1, SECONDS));
```

可以看到，sample() 的行为有一些很微妙的地方。与其依赖于文档和手工验证的理解，还不如进行一些自动化测试。7.2.1 节将会讨论对时间敏感的操作符，例如 sample()。

在 RxJava 中，sample() 有一个别名，即 throttleLast()。与之对应的还有 throttleFirst() 操作符，它会发布每个阶段中的第一个事件。所以，使用 throttleFirst() 来替换 sample()，预期会得到如下的结果。

```
Observable<String> names = Observable
    .just("Mary", "Patricia", "Linda",
        "Barbara",
        "Elizabeth", "Jennifer", "Maria", "Susan",
        "Margaret", "Dorothy");

Observable<Long> absoluteDelayMillis = Observable
    .just(0.1, 0.6, 0.9,
        1.1,
        3.3, 3.4, 3.5, 3.6,
        4.4, 4.8)
    .map(d -> (long)(d * 1_000));
```

```
//...

delayedNames
    .throttleFirst(1, SECONDS)
    .subscribe(System.out::println);
```

输出如下所示。

```
Mary
Barbara
Elizabeth
Margaret
```

与 `sample()` 类似（即 `throttleLast()`），如果在 Barbara 和 Elizabeth 之间没有新的人名出现，那么 `throttleFirst()` 不会发布任何的事件。

6.1.2　将事件缓冲至列表中

在 RxJava 提供的内置操作符中，缓冲和窗口移动（moving window）是最令人兴奋的。它们都会通过一个窗口（window）遍历整个输入流，这个窗口会捕获多个连续的元素并推动流程向前运行。另一方面，它们允许批处理上游源中的值，从而实现更高效的处理。在实践中，它们是灵活而且用途广泛的工具，可以在运行期对数据进行各种聚合。

`buffer()` 操作符会将一批事件实时聚合到一个 `List` 中。但是，与 `toList()` 操作符不同，`buffer()` 发布的列表会对随后出现的事件进行分组，而不是将所有的事件放到同一个列表中（如 `toList()`）。`buffer()` 最简单的形式就是将上游 `Observable` 中的值分组到相同大小的列表中。

```
Observable
    .range(1, 7)  //1, 2, 3, ... 7
    .buffer(3)
    .subscribe((List<Integer> list) -> {
            System.out.println(list);
        }
    );
```

当然，`subscribe(System.out::println)` 也可以正常运行，这里出于教学的目的，保留了类型的信息。下面的输出展示了 `buffer(3)` 操作符发布的三个事件。

```
[1, 2, 3]
[4, 5, 6]
[7]
```

`buffer()` 会不断接收上游的事件，并在内部对它们进行缓冲（因此得名），直到缓冲区的大小达到 3，此时整个缓冲区（`List<Integer>`）会被推送至下游。完成通知出现时，如果内部缓冲区不为空（但是还没有达到 3），它依然会被推送至下游。这就是为什么上面输出的最后一行是只包含一个元素的列表。

使用 `buffer(int)` 操作符，可以将细粒度的事件替换为数量更少但规模较大的批处理。例如，如果你想要减少数据库的负载，那么就可以将每个事件单独存储的方案替换为批量存储。

```
interface Repository {
    void store(Record record);
    void storeAll(List<Record> records);
}

//...

Observable<Record> events = //...

events
        .subscribe(repository::store);
//vs.
events
        .buffer(10)
        .subscribe(repository::storeAll);
```

后面的订阅调用了 Repository 中的 storeAll，一次性保存了 10 个元素。这种方式可能会提高应用程序的吞吐量。

buffer() 有很多的重载变种。buffer() 将列表推送至下游时，一个稍微复杂的版本允许配置内部缓冲区中最旧值的丢弃数量。这听起来非常复杂，用更基础的术语就是：它实现了通过指定大小的移动窗口来查看事件流。

```
Observable
        .range(1, 7)
        .buffer(3, 1)
        .subscribe(System.out::println);
```

这样，样例将会产生多个重叠的列表。

```
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
[4, 5, 6]
[5, 6, 7]
[6, 7]
[7]
```

如果想要计算某些时序数据的移动平均值（moving average），那么我们可以使用 buffer(N, 1) 变种形式。如下的样例代码将会按照正态分布生成 1000 个随机值。随后，取一个 100 个元素组成的滑动窗口（每次前进一个元素），计算这个窗口的平均值。[1] 亲自运行这个程序，并观察移动平均值，你会发现它比随机无序值要平滑得多。

```
import java.util.Random;
import java.util.stream.Collectors;

//...

Random random = new Random();
Observable
        .defer(() -> just(random.nextGaussian()))
```

注 1：注意这不是最高效的算法，因为该算法多次添加相同的数字。

```
        .repeat(1000)
        .buffer(100, 1)
        .map(this::averageOfList)
        .subscribe(System.out::println);

//...

private double averageOfList(List<Double> list) {
    return list
            .stream()
            .collect(Collectors.averagingDouble(x -> x));
}
```

你可以将 buffer(N) 视为与 buffer(N, N) 等价。最简单的 buffer() 形式会在整个缓冲区充满之后丢弃它。有意思的是，buffer(int, int) 的第二个参数（用来指定向下游推送时要跳过元素的数量）可以大于第一个参数，这实际上跳过了一些元素！

```
Observable<List<Integer>> odd = Observable
        .range(1, 7)
        .buffer(1, 2);
odd.subscribe(System.out::println);
```

这种设置会转发第一个元素，但随后会跳过两个元素，也就是第一个和第二个元素。然后循环这个过程：buffer() 转发第三个元素，但是会跳过的第三个和第四个元素。它的输出将会是：[1] [3] [5] [7]。注意，odd Observable 中的每个元素实际上都是一个单元素列表。可以使用 flatMap() 或 flatMapIterable() 把它重新变成 Observable<Integer>。

```
Observable<Integer> odd = Observable
        .range(1, 7)
        .buffer(1, 2)
        .flatMapIterable(list -> list);
```

flatMapIterable() 预期接收一个函数，该函数会将流（由单个元素组成的 List<Integer>）中的每个元素转换到一个 List 中。在这里，恒等转换（list -> list）就达到要求了。

基于时间阶段进行缓冲

实际上，buffer() 是一个用途广泛的操作符家族。buffer() 的另一个变种形式能够根据时间阶段批量处理上游的事件，而不是根据大小来处理（在这种情况下，每个批次的事件数量是相同的）。throttleFirst() 和 throttleLast() 能够在给定时间段内分别获取第一个和最后一个元素，但是 buffer 的一个重载版本能够在每个时间段批量处理所有的事件。接下来回到人名样例。

```
Observable<String> names = just(
        "Mary",     "Patricia", "Linda", "Barbara", "Elizabeth",
        "Jennifer", "Maria",    "Susan", "Margaret","Dorothy");
Observable<Long> absoluteDelays = just(
    0.1, 0.6, 0.9, 1.1, 3.3,
    3.4, 3.5, 3.6, 4.4, 4.8
).map(d -> (long) (d * 1_000));

Observable<String> delayedNames = Observable.zip(names,
        absoluteDelays,
```

```
        (n, d) -> just(n).delay(d, MILLISECONDS)
).flatMap(o -> o);

delayedNames
        .buffer(1, SECONDS)
        .subscribe(System.out::println);
```

buffer() 的重载版本接收一个时间阶段（在上例中就是 1 秒），它会聚集该时间段内所有的上游事件。因此，buffer() 会收集第一个时间段、第二个时间段内发生的所有事件，以此类推。

```
[Mary, Patricia, Linda]
[Barbara]
[]
[Elizabeth, Jennifer, Maria, Susan]
[Margaret, Dorothy]
```

第三个 List<String> 为空，因为在这个时间段内没有出现事件。buffer() 的一个用例就是统计每个时间段内事件的数量，如计算 1 秒内关键事件的数量。

```
Observable<KeyEvent> keyEvents = //...

Observable<Integer> eventPerSecond = keyEvents
        .buffer(1, SECONDS)
        .map(List::size);
```

比较好的一点是，如果在 1 秒内没有事件，样例将会生成一个空的列表，这样，在度量中就不会出现空白。但是，这并不是最有效的方式，稍后将会介绍 window() 操作符。

buffer() 方法最全面的重载形式可以控制该操作符何时开始缓冲事件，以及何时将缓冲区的内容刷新到下游。换句话说，你可以选择在哪个时间段对上游的事件进行分组。假设你要监控一些频繁推送遥测数据的工业设备，这种场景下的数据量会非常大，为了节省计算能力，样例可以只查看特定的采样数据。算法如下所示。

- 在业务时间（9：00 至 17：00），每秒获取 100 毫秒的快照（大约处理 10% 的数据）。
- 在业务时间之外，每 5 秒获取 200 毫秒的快照（4%）。

换句话说，每秒（或每 5 秒）缓冲 100 毫秒（或者相应的 200 毫秒）内所有的事件，并将这个时间段内的所有事件形成的列表发布出去。如果你看到完整的例子，就会更加清楚了。首先需要一个 Observable，开始进行缓冲（分组）上游事件时，它能随意发布一个值。这个 Observable 可以放入任何类型的值，它的内容并不重要，重要的是时间。实际上，返回 java.time 包中的 Duration 只是一个巧合，RxJava 并没有以任何形式使用这个值。

```
Observable<Duration> insideBusinessHours = Observable
    .interval(1, SECONDS)
    .filter(x -> isBusinessHour())
    .map(x -> Duration.ofMillis(100));
Observable<Duration> outsideBusinessHours = Observable
    .interval(5, SECONDS)
    .filter(x -> !isBusinessHour())
    .map(x -> Duration.ofMillis(200));
```

```
Observable<Duration> openings = Observable.merge(
    insideBusinessHours, outsideBusinessHours);
```

首先，使用 interval() 生成按秒执行的计时节拍，不过这里排除非业务时间。通过这种方式，在 9：00 和 17：00 之间的每秒生成了一个稳定的时钟节拍。回忆一下 interval() 操作符，它返回的是不断递增的 Long 类型自然数，但是，我们并不需要这样的值，因此将其替换成了 100 毫秒的固定间隔。与之完全类似的后续代码，在 17：00 和 9：00 之间每 5 秒生成一个值的稳定流。如果你关心如何实现 isBusinessHour()，其实它用到了 java.time 包。

```
private static final LocalTime BUSINESS_START = LocalTime.of(9, 0);
private static final LocalTime BUSINESS_END = LocalTime.of(17, 0);

private boolean isBusinessHour() {
    ZoneId zone = ZoneId.of("Europe/Warsaw");
    ZonedDateTime zdt = ZonedDateTime.now(zone);
    LocalTime localTime = zdt.toLocalTime();
    return !localTime.isBefore(BUSINESS_START)
        && !localTime.isAfter(BUSINESS_END);
}
```

openings 流会将 insideBusinessHours 和 outsideBusinessHours 流**合并**（merge）在一起。它基本上就是一个定时器，指示 buffer() 操作符何时从上游中采样，而不是将其丢弃。openings 流中发布什么样的值完全没有关系。但是，必须还要指定何时停止聚合（缓冲）事件，并将其以一个 List 的形式推送至下游。最显而易见的解决方案就是将 openings 流中的每个事件作为停止当前批处理的一个信号，样例将缓冲的事件发布至下游，并开始新的批处理。

```
Observable<TeleData> upstream = //...

Observable<List<TeleData>> samples = upstream
    .buffer(openings);
```

注意样例是如何将精心设计的 openings 传递给 buffer() 操作符的。上述的代码片段切分了由 TeleData 值组成的 upstream。openings 流的时钟节拍会批量处理来自 upstream 的事件。在业务时间内，每秒会生成一个批量数据；而在业务时间之外，样例会按照 5 秒一组的节奏对值进行批量分组。在这个版本中有一点很重要，来自 upstream 的所有事件都会被保留下来，因为它们要么属于这个批次，要么属于另外一个批次。不过，重载版本的 buffer() 可以标记某个批处理何时结束。

```
Observable<List<TeleData>> samples = upstream
    .buffer(
        openings,
        duration -> empty()
            .delay(duration.toMillis(), MILLISECONDS));
```

之前提到过 openings，它是一个 Observable<Duration>，但是 openings 流中的事件实际值并不重要。RxJava 只是使用这个事件开始缓冲 TeleData 实例。只不过这一次样例能够完全控制何时开始缓冲，以及何时将缓冲的内容发布出去。第二个参数是一个 Observable，

它必须在我们想要停止采样的时候完成。第二个流的完成标志着给定批次的结束。这里需要仔细观察：我们想要开始一个新的批次时，openings 流会发布一个事件。对于 openings 发布的每个事件，样例都会返回一个新的 Observable，它应该在未来的某个时间点完成。例如，openings 发布一个值为 Duation.ofMillis(100) 的事件时，样例将其转换为一个 Observable，它会在给定的批次结束时完成，在这里也就是 100 毫秒之后。注意，在连续的批次中，有些事件可能会被丢弃或重复出现。如果第二个 Observable，也就是负责标记给定批次完成的 Observable，出现在下一个批次的开始事件之前，那么在这个时间差之内的事件将会被 buffer() 丢弃。具体到这里：每秒（在业务时间之外是其他间隔秒数）都会缓冲事件，但是缓冲区会在 100 毫秒（或者 200 毫秒，视情况而定）之后关闭，缓冲内容会被转发至下游。大多数事件都会落在缓冲时段中，因此会被丢弃。

buffer() 操作符非常灵活和复杂。请你用它做一些试验性的尝试并确保理解了上述样例。它能够非常智能地批量处理来自上游的事件，从而实现分组、采样和窗口移动的功能。但是，buffer() 关闭当前缓冲时需要创建一个中间的 List，然后才能将其传递到下游，这可能会给垃圾回收和内存使用带来不必要的压力（参见 8.6 节）。因此，window() 操作符应运而生。

6.1.3 窗口移动

我们在使用 buffer() 时需要不断地创建 List 实例。既然如此，为什么要创建这些中间的 List，而不是以某种方式直接在运行时消费事件呢？这就是 window() 操作符的用武之地了。如果可能，我们应该优先使用 window()，而不是 buffer() 操作符，因为后者在内存消耗方面的行为更难预料。window() 操作符与 buffer() 非常类似：它有类似的重载版本，包括如下功能的重载形式。

- 接收 int 类型，将来自源的事件分组到固定大小的列表中。
- 接收时间单元，将事件按照固定的时间阶段进行分组。
- 接收自定义的 Observable，用来标记每个批次的开始和结束。

那么，差异何在呢？前面提到过记录指定源每秒产生事件数量的样例，接下来重新看一下这个样例。

```
Observable<KeyEvent> keyEvents = //...

Observable<Integer> eventPerSecond = keyEvents
    .buffer(1, SECONDS)
    .map(List::size);
```

样例批量获取 Observable<KeyEvent> 每秒产生的事件，并将其放到 Observable<List<KeyEvent>> 中。下一步通过 map 将 List 映射为它的 size。这相当浪费，每秒产生的事件数量非常大就更是如此。

```
Observable<Observable<KeyEvent>> windows = keyEvents.window(1, SECONDS);
Observable<Integer> eventPerSecond = windows
    .flatMap(eventsInSecond -> eventsInSecond.count());
```

与 buffer() 不同，window() 返回的是一个 Observable<Observable<KeyEvent>>。这里接收

的不是固定的列表，这类列表每个存放一个批次（或缓冲），接收的是由一系列流组成的流。每次新的批次开始的时候（在上述样例中就是每一秒），在外层流中都会出现一个新的 Observable<KeyEvent> 值。这些内部流可以进一步转换，但是为了避免双重包装，我们使用 flatMap()。flatMap() 会接收每个缓冲（Observable<KeyEvent>）作为参数，并预期返回另外一个 Observable。count() 操作符（参见 3.4 节）会将 Observable<T> 转换成一个 Observable<Integer>，它只发布一个条目，这个条目代表了原始 Observable 中的事件数量。因此，对于样例生成的所有一秒批次，就是该秒内发生的事件数量。但是，这里没有内部的缓冲，count() 操作符会在事件通过时动态地记录它们的数量。

6.1.4 借助debounce()跳过陈旧的事件

buffer() 和 window() 会将几个事件组合在一起，从而对它们进行批量处理。sample() 偶尔会随机选取一个事件。这些操作符并没有考虑事件之间的时间间隔。但是在很多场景下，如果某个事件紧跟着另外一个事件，那么前者可能就会被丢弃。例如，假设有一个股票价格形成的流从交易平台流出。

```
Observable<BigDecimal> prices = tradingPlatform.pricesOf("NFLX");
Observable<BigDecimal> debounced = prices.debounce(100, MILLISECONDS);
```

debounce()（即 throttleWithTimeout()）将会丢弃紧随另一个事件的所有事件。换句话说，如果给定事件与另一个事件的间隔超出了时间窗口，那么这个值才会发布出去。在前面的样例中，每当 NFLX 股票的价格发生变化，prices 流就会推送该价格。有时候，股票价格的变动会非常频繁，每秒就会有十多次变化。对于价格的每次变动，我们都会运行一些计算，而这些计算需要耗费一定的时间才能完成。但是，如果有新价格出现，之前的计算就没有意义了，要根据新价格从头开始计算。因此，如果某些事件紧随一个新事件（或者说被新事件抑制了），我们可能想要将其丢弃。

debounce() 会等待一小段时间（在前面的样例中也就是 100 毫秒），以防第二个事件会随后出现。这个过程会不断重复。所以，如果第二个事件出现时，距离第一个事件不超过 100 毫秒，RxJava 将会延迟它的发布，并期待第三个事件的出现。这里依然可以灵活控制每个事件的等待时间。例如，如果股票价格的更新间隔小于 100 毫秒，我们可能想忽略它。但是，如果价格超过了 150 美元，我们可能就想立刻将这样的更新更快地转发至下游。有些类型的事件可能需要立即处理，比如它们代表着巨大的市场机会。使用重载版本的 debounce()，可以很容易地实现这种需求。

```
prices
    .debounce(x -> {
        boolean goodPrice = x.compareTo(BigDecimal.valueOf(150)) > 0;
        return Observable
            .empty()
            .delay(goodPrice? 10 : 100, MILLISECONDS);
    })
```

对于价格 x 的每次更新，样例都会应用一个复杂的逻辑（大于 150 美元），以判断价格是否合适。然后，对于每次这样的变更，样例都会返回唯一的 Observable。这个 Observable 是空的，不需要发布任何条目，重要的是它何时能够完成。对于合适的价格，这个

Observable 会在 10 毫秒后发布一个完成通知；而对于其他的价格，它会在 100 毫秒之后完成。debounce() 对接收到的每个事件都订阅该 Observable，并等待它的完成。如果它首先完成，这个事件会发布到下游。否则，如果更新的上游事件同时出现，这个循环将会重复执行。

在样例中，价格 x 的值为 140 美元时，debounce() 会创建一个新的 Observable，根据传递的表达式，这个 Observable 会延迟 100 毫秒完成。如果在这个事件完成之前，没有其他事件出现，这个 140 美元的事件就会转发到下游。但是，假设价格 x 更新为 151 美元。这次，debounce() 要求提供 Observable（在 API 中，名为 debounceSelector）时，返回了一个完成时间要短得多的流，10 毫秒就可以完成。所以，如果出现了合适的价格（大于 150 美元），只需等 10 毫秒就可以判断是否出现后续的更新。如果你依然很难理解 debounce() 的工作原理，可以尝试运行下面的股票价格模拟程序。

```
Observable<BigDecimal> pricesOf(String ticker) {
    return Observable
            .interval(50, MILLISECONDS)
            .flatMap(this::randomDelay)
            .map(this::randomStockPrice)
            .map(BigDecimal::valueOf);
}

Observable<Long> randomDelay(long x) {
    return Observable
            .just(x)
            .delay((long) (Math.random() * 100), MILLISECONDS);
}

double randomStockPrice(long x) {
    return 100 + Math.random() * 10 +
            (Math.sin(x / 100.0)) * 60.0;
}
```

上述样例很漂亮地将多个流组合在了一起。首先，生成一个 long 类型的序列，它们按照固定的 50 毫秒间隔发布。然后，每个事件独立地延迟 0~100 毫秒。最后，将无限递增的 long 类型数字转换为随机抖动的正弦曲线（使用 Math.sin()）。这模拟了股票价格随时间的波动。如果使用 debounce() 运行这个流，你会发现如果价格比较低，事件就不会频繁出现，因为要等待后续的事件 100 毫秒，而后续事件经常发生。但是，如果价格超过了 150 美元，debounce() 的容量就降低到了 10 毫秒，这样，每个合适价格的更新都会转发至下游。

避免在debounce()中出现饥饿现象

很容易想象在某种情景下，debounce() 操作符会阻止所有事件的发布，因为这些事件出现得过于频繁，没有片刻停歇。

```
Observable
    .interval(99, MILLISECONDS)
    .debounce(100, MILLISECONDS)
```

这样的源不会发布任何的事件，因为 debounce() 会等待 100 毫秒，以确保没有新的事件发生。令人遗憾的是，在距离这个超时还有 1 毫秒的时候，新事件就出现了，这样会重新启动 debounce 的计时器。这导致，如果一个 Observable 生成事件过于频繁，我们可能永远都看不

到任何的事件！你可以将其称为一种特性，但是在实践中，即便出现了事件洪峰，我们依然想要不时地看到**一些**事件。为了避免上述情况的出现，需要采取一些有创造性的做法。

首先，样例必须识别是否存在长时间没有新事件出现的情况。7.1.3 节会介绍 `timeout()` 操作符，其实这部分非常简单。

```
Observable
        .interval(99, MILLISECONDS)
        .debounce(100, MILLISECONDS)
        .timeout(1, SECONDS);
```

现在，至少能够得到一个异常信号，提示上游源是空闲的。令人惊讶的是，事实恰好相反。上游的 `interval()` 操作符生成事件过于频繁，所以 `debounce()` 并没有将它们传递到下游，我们在这里有点误入歧途了。如果事件出现得过于频繁，样例会将它们临时保留并停歇片刻。但是如果这个沉默等待的时间太长（超过 1 秒），程序就会失败并抛出 `TimeoutException`。与其让程序永远失败，我们更希望看到上游 `Observable` 中的任意一个值并继续运行。任务的第一部分非常简单。

```
ConnectableObservable<Long> upstream = Observable
        .interval(99, MILLISECONDS)
        .publish();
upstream
        .debounce(100, MILLISECONDS)
        .timeout(1, SECONDS, upstream.take(1));
upstream.connect();
```

`timeout()` 操作符有一个重载版本，能够在超时的情况下接收一个备用的 `Observable`。但是，这里有一个小缺陷。如果出现超时，这里只会从上游中获取第一个条目就结束了。我们真正想要达到的效果是继续发布 `upstream` 中的事件，并依然支持 `debounce()`。

> `ConnectableObservable`
>
> 在这里需要将 `ConnectableObservable` 与 `publish()`、`connect()` 组合使用，从而将 cold 类型的 `Observable.interval()` 转换成 hot 类型（参见 2.4.4 节，了解 `interval()` 为何是 cold 类型以及这意味着什么）。通过调用 `publish()` 和 `connect()`（参见 2.7.2 节），强迫 `interval()` 操作符立即生成事件，即便无人订阅。这意味着，如果几秒之后再订阅这个 `Observable`，它就会从中间开始接收事件，所有的订阅者会在同一时刻得到相同的事件。默认情况下，`interval()` 是 cold 类型的 `Observable`，所以每个订阅者不管何时订阅，都会从头开始接收事件。

如下的另外一种方式看上去会更好一些。

```
upstream
    .debounce(100, MILLISECONDS)
    .timeout(1, SECONDS, upstream
        .take(1)
        .concatWith(
            upstream.debounce(100, MILLISECONDS)))
```

乍看上去，它是没有问题的。最初的源在应用 debounce() 操作符之后有一个超时操作符。出现超时的时候，样例会发布遇到的第一个条目，然后继续使用相同的源，这也是通过 debounce() 操作符实现的。但是，如果出现第一次超时，我们切换到了备用 Observable 上，而这个 Observable 是没有使用 timeout() 操作符的。有一种快速、混乱且短视的修正方案，如下所示。

```
upstream
    .debounce(100, MILLISECONDS)
    .timeout(1, SECONDS, upstream
        .take(1)
        .concatWith(
            upstream
                .debounce(100, MILLISECONDS)
                .timeout(1, SECONDS, upstream)))
```

以上代码再次忘记了在内层 timeout() 操作符中添加备用 Observable。足够了，你应该已经注意到了递归模式。与其重复使用相同形式的 upstream → debounce → timeout() → upstream →⋯，其实可以使用递归。

```
import static rx.Observable.defer;


Observable<Long> timedDebounce(Observable<Long> upstream) {
    Observable<Long> onTimeout = upstream
            .take(1)
            .concatWith(defer(() -> timedDebounce(upstream)));
    return upstream
            .debounce(100, MILLISECONDS)
            .timeout(1, SECONDS, onTimeout);
}
```

在 timedDebounce 中，备用 Observable 的 onTimeout 比较复杂。它声明首先要从 upstream（这是最初的源）中获取第一个采样事件，随后递归调用 timedDebounce() 方法。样例必须使用 defer() 来避免无穷递归。timedDebounce() 的剩余内容基本上就是基于最初的上游源，应用 debounce() 操作符并添加备用 onTimeout。这个备用 Observable 做了完全相同的事情：递归地应用 debounce()、添加 timeout() 和备用 Observable。

如果你觉得上述代码难以理解，不必感到沮丧。这是一个非常复杂的样例，展现了流组合、延迟执行和递归的威力。我们很少需要这么高的复杂程度，但是掌握了它的运行原理之后，你会很有成就感。请仔细研读这段代码，并观察微小的变化会如何极大地影响流之间的交互方式。

6.2　回压

对于构建健壮和即时响应的应用程序，回压（backpressure）是至关重要的。在本质上，它是一个消费者到生产者的反馈通道。消费者能够在一定程度上控制它随时能够处理多少数据。这样在高负载的情况下，消费者或消息中间件不会变得饱和而无响应。相反，它们可以请求更少的信息，由生产者决定如何放慢速度。

在所有通过消息传递（或事件）进行数据交换的系统中，都可能会出现消费者的节奏跟不上生产者的问题。根据底层实现的不同，这种问题可能会表现为不同的形式。如果通信通道以某种方式同步生产者和消费者（比如使用 `ArrayBlockingQueue`），当消费者跟不上负载的变化时，生产者就会节流（阻塞）。这将会导致生产者和消费者之间产生耦合，而两者本应是完全独立的。消息传递一般意味着异步处理，如果生产者必须要等待消费者，那么这种假设就难以成立了。更糟糕的是，在更高层级上来看，这个生产者可能还是其他生产者的消费者，这会导致延迟的级联增加。

相反，即使两者的沟通介质是无界的，它依然会受到无法控制的因素的制约。像 `LinkedBlockingQueue` 这样的无限队列，允许生产者在不阻塞的前提下存放大量超出消费者处理能力的消息。换句话说，在 `LinkedBlockingQueue` 没有消耗掉所有的内存并让应用程序崩溃之前，其实可以一直往里面堆放消息。如果中间介质是持久化的，比如 JMS 的消息代理（message broker），同样的问题会以磁盘空间的形式表现出来，但是这种情况发生的可能性比较低。更加常见的情况是，消息中间件难以管理数千条甚至数百万条未使用的消息。有些特殊的代理（broker），比如 Kafka，在技术上能够存储数亿条消息，直到处理速度落后的消费者获取到这些消息为止。但是，这会导致延迟的大幅增加，这里的延迟指的是消息的生产和消费的间隔时间。

尽管通常认为消息驱动的系统更加健壮和可扩展，但是生产者过于活跃的问题依然存在。不过，为了解决这个集成相关的问题，已经有了一些相关的尝试。在 RxJava 中，采样、节流（使用 `sample()` 等）以及批处理（使用 `window()` 和 `buffer()`）是手动减少生产者负载的方法。`Observable` 生成的事件数量超过消费能力时，可以使用采样或批处理增加订阅的吞吐量。然而，我们需要更加健壮和系统化的方案，所以反应式流（Reactive Stream）应运而生。这是一组很小的接口和语义集，旨在正式解决这个问题，并为生产者和消费者的协调提供一个系统化的算法，即**回压**。

回压是一个简单的协议，它允许消费者请求一次可以消费多少数据，实际上是提供了一条面向生产者的反馈通道。生产者接收来自消费者的请求，避免消息溢出。当然，这种算法只能用于生产者能够对自己进行节流的场景，比如它们使用静态集合作为支撑，或者数据源通过类似 `Iterator` 这样的形式进行拉取。如果生产者无法控制生成数据的频率（数据源是外部的，或者是 hot 类型的），那么回压也提供不了太大的帮助。

6.2.1　RxJava中的回压

尽管反应式流以技术中立的方式解决了一个普遍存在的问题，但是我们依然要关注 RxJava，关注它是如何解决回压问题的。本章将会使用小餐馆中刷盘子的样例，这里将盘子建模为具有一个标识符的大对象。

```
class Dish {
    private final byte[] oneKb = new byte[1_024];
    private final int id;

    Dish(int id) {
        this.id = id;
        System.out.println("Created: " + id);
    }
```

```
        public String toString() {
            return String.valueOf(id);
        }
    }
```

这里的 oneKb 缓冲区只是为了模拟一些额外的内存占用。盘子通过服务员传递给厨房，我们将其建模为 Observable。

```
Observable<Dish> dishes = Observable
    .range(1, 1_000_000_000)
    .map(Dish::new);
```

range() 操作符会尽可能快地生成新的值。如果刷盘子需要一些时间，而且明显要比生成的节奏慢，将会发生什么呢？

```
Observable
    .range(1, 1_000_000_000)
    .map(Dish::new)
    .subscribe(x -> {
        System.out.println("Washing: " + x);
        sleepMillis(50);
    });
```

有点令人意外，情况并不糟糕。如果你研读一下输出，就会发现 range() 与订阅完全匹配。

```
Created: 1
Washing: 1
Created: 2
Washing: 2
Created: 3
Washing: 3
...
Created: 110
Washing: 110
...
```

这应该并不意外。range() 操作符默认并不是异步的，所以它生成的每个条目会直接在当前线程的上下文中传递给 Subscriber。如果 Subscriber 比较缓慢，就阻止了 Observable 生成更多的元素。在前一个条目完成之前，range() 无法调用 Subscriber 的 onNext()。这可能是因为生产者和消费者在同一个线程中运行，二者透明地耦合在一起。在某种程度上可以说，它们之间有一个隐式的队列，这个队列的最大容量就是 1。这是我们始料未及的**会合**（rendezvous）算法。假设饭店的服务员在当前的盘子洗完之前，无法将新的待洗盘子送到厨房，而且服务员必须要站在那里等待盘子洗完，此时顾客就没人招待了。这些顾客没有得到招待，新顾客也没有办法进入餐厅。这样，阻塞式组件让整个系统陷入了停顿。但是，在实际中，生产者和消费者之间会有一个线程边界：Observable 在一个线程中生产事件，而 Subscriber 在另一个线程中消费事件。

```
dishes
    .observeOn(Schedulers.io())
    .subscribe(x -> {
        System.out.println("Washing: " + x);
```

```
        sleepMillis(50);
    });
```

先暂停一下，想想如果不实际编译和运行代码会发生什么。有人可能认为会产生灾难性的结果，因为来自 range() 操作符的 dishes 会非常快地生成事件，而 Subscriber 消费的速度非常慢，每秒只能消费 20 个盘子。observeOn() 操作符能够持续快速地消费事件，但是 Subscriber 消费事件的方式太慢了。所以你可能会得出结论：OutOfMemoryError 是难以避免的，依据就是未处理的事件会在某处堆积。幸而，回压扭转了败局，而且 RxJava 在一定程度上保护了我们。程序的输出有点出人意料。

```
Created: 1
Created: 2
Created: 3
...
Created: 128

Washing: 1
Washing: 2
...
Washing: 128

Created: 129
...
Created: 223
Created: 224

Washing: 129
Washing: 130
...
```

首先，range() 几乎在瞬间生成了一批盘子（128 个）。随后，是比较慢的逐个刷盘子的过程。在某种程度上来说，此时 range() 操作符空闲了下来。这 128 个盘子中的最后一个清洗完毕之后，range() 又生成了一批盘子（96 个），在之后又是缓慢的清洗过程。[2] 显然，这里必须存在某种智能的机制避免 range() 生成太多的事件，具体生成事件的数量由订阅者控制。如果你没有看到这种机制是在哪里发挥作用的，那尝试一下自行实现 range()。

```
Observable<Integer> myRange(int from, int count) {
    return Observable.create(subscriber -> {
        int i = from;
        while (i < from + count) {
            if (!subscriber.isUnsubscribed()) {
                subscriber.onNext(i++);
            } else {
                return;
            }
        }
        subscriber.onCompleted();
    });
}
```

接下来，在相同的样例中结合 observeOn() 来使用 myRange()。

注 2：不必过分关注盘子具体的数量，关键的是请求批量事件的周期性。

```
myRange(1, 1_000_000_000)
        .map(Dish::new)
        .observeOn(Schedulers.io())
        .subscribe(x -> {
                    System.out.println("Washing: " + x);
                    sleepMillis(50);
                },
                Throwable::printStackTrace
        );
```

最终的结果是灾难性的，一个盘子都没有清洗。

```
Created: 1
Created: 2
Created: 3
...
Created: 7177
Created: 7178

rx.exceptions.MissingBackpressureException
    at rx.internal.operators...
    at rx.internal.operators...
```

随后将会阐述 `MissingBackpressureException`。现在，你能够确信自定义的 `range()` 实现缺少了某种后台机制。

6.2.2 内置的回压

我们在前几章观察了事件是如何从源 `Observable` 流到下游的，这个过程会经历一系列的操作符，然后到达一个 `Subscriber`。订阅请求并没有任何的反馈通道。样例调用 `subscribe()` 的时候（在某种意义上是向上传递），所有的事件和通知都会向下传递，中间没有任何明显的反馈回路。缺少反馈可能会导致生产者（最高的 `Observable`）发布大量的事件，这超出了订阅者的处理能力。结果就是，应用程序可能会因为 `OutOfMemoryError` 而崩溃，或者再好一点，但也存在很大的潜在威胁。

回压机制能够让终端订阅者和中间操作符从生产者那里只请求特定数量的事件。默认情况下，上游 cold 类型的 `Observable` 会尽快生成事件。但是，下游发起这类请求时，它应该按照某种方式“慢下来”并生成符合请求数量的事件。这样，在 `observeOn()` 中就出现了 128 这个神奇的数字。但是，先看一下最终的订阅者是如何控制回压的。

订阅时，样例可以实现 `onNext()`、`onCompleted()` 和 `onError()`（参见 2.2 节）。实际上还有另外一个回调方法：`onStart()`。

```
Observable
    .range(1, 10)
    .subscribe(new Subscriber<Integer>() {

        @Override
        public void onStart() {
            request(3);
        }
```

```
        //随后是onNext，onCompleted，onError……
    });
```

RxJava 调用 onStart() 的时机和我们预想的一样，也就是在事件或通知传递到 Subscriber 之前。在技术上，你可以使用 Subscriber 的构造器，但是对于 Java 中的匿名内部类，构造器看上去有些怪异。

```
.subscribe(new Subscriber<Integer>() {

    {{
        request(3);
    }}

    //随后是onNext，onCompleted，onError……
});
```

有点跑题了。在 Subscriber 中调用 request(3)，会向上游的源指明我们最开始想要接收的条目数量。如果完全忽略对它的调用（或者调用 request(Long.MAX_VALUE)），那么就等价于请求尽可能多的事件。这就是尽早调用 request() 原因，如果流已经开始发布事件，就无法减少要求的数量了。但是，如果只请求三个事件，range() 操作符会在推送完 1、2 和 3 之后暂时停止发布事件。onNext() 回调方法会被调用三次，之后，尽管 range() 操作符还没有完成，onNext() 也不会再调用了。但是，Subscriber 完全可以控制想要接收多少数据。例如，我们想要分别请求事件。

```
Observable
    .range(1, 10)
    .subscribe(new Subscriber<Integer>() {

        @Override
        public void onStart() {
            request(1);
        }

        @Override
        public void onNext(Integer integer) {
            request(1);
            log.info("Next {}", integer);
        }

        //onCompleted, onError...
    });
```

这个样例有点牵强，它的行为与没有任何回压功能的普通 Subscriber 完全一样。但是，它阐明了如何使用回压。你可以想象一个 Subscriber，它可以预先缓冲一定数量的事件，然后在方便的时候再进行批量请求。尽管 Subscriber 处于空闲状态，但它依然可能会在接收更多事件之前等待一会儿，比如为了减轻对下游依赖的压力。在餐厅样例中，服务员是一个不断推送新的脏盘子的 Observable<Dish>，而 request(N) 表明厨房工作人员准备清洗一定数量的盘子。如果没有厨房工作人员的请求，一个称职的服务员是不应该将新的脏盘子传送进来的。

也就是说，在客户端代码中直接调用 request(N) 是比较少见的。更常见的情况是，放置在源和最终的 Subscriber 之间的各种操作符，利用回压功能控制有多少数据流经管道。例如，observeOn() 必须要订阅上游的 Observable，并在一个特定的 Scheduler（比如 io()）中调度接收到的事件。但是，如果上游生成事件的速度超过了底层 Scheduler 和 Subscriber 的处理速度，那又会怎样呢？ observeOn() 创建的 Subscriber 是支持回压的，它最初只会请求 128 个值。[3] 上游的 Observable 能够理解回压机制，所以只发布给定数量的事件，然后就会处于空闲状态了，这就是样例中 range() 做的事情。当 observeOn() 发现当前批次的事件已经被下游 Subscriber 处理完毕，它会请求更多的事件。这样尽管跨越了线程边界，而且生产者和消费者端都是异步的，消费者依然不会被事件洪峰击溃。

observeOn() 并不是唯一对回压友好的操作符，很多操作符也充分利用了回压功能。比如，zip() 只会缓冲来自底层 Observable 的固定数量的事件。幸亏有了这个特性，当压缩的流中只有一个非常活跃的时候，zip() 也不会受到影响。相同的逻辑适用于之前用过的大多数操作符。

6.2.3 Producer与缺失回压场景

在自定义的 range() 实现中，我们已经见到过 MissingBackpressureException 了。它到底意味着什么，又该怎样解读这个异常呢？假设有个 Subscriber（可能是自己创建的，但更常见的是由某个操作符创建的），能够精确知道它想要接收多少个条目，如 buffer(N) 或 take(N)。这类操作符的另外一个例子是 observeOn()。它在某些方面必须非常严格，如果上游 Observable 基于某种原因推送了更多的条目，那么 observeOn() 内部的缓冲将会溢出，并发出 MissingBackpressureException 这样的信号。话又说回来，上游 Observable 推送的条目为什么会超出请求的数量呢？这只是因为它忽略了对 request() 的调用。我们回顾一下简单 range() 的重新实现，如下所示。

```
Observable<Integer> myRange(int from, int count) {
    return Observable.create(subscriber -> {
        int i = from;
        while (i < from + count) {
            if (!subscriber.isUnsubscribed()) {
                subscriber.onNext(i++);
            } else {
                return;
            }
        }
        subscriber.onCompleted();

    });
}
```

停止的唯一办法是取消订阅，但是我们并不想取消订阅，只是想要让它的速度减慢一些。下游的操作符精确地知道它们想要接收多少事件，但是源忽略了这些请求。用于表示请求的事件数量的底层机制是通过 rx.Producer 实现的。这个接口会在 create() 的时候插进来。回忆一下，每次有人订阅 Observable，OnSubscribeRange 回调都会执行。通常情况下，

注 3：你可以通过 rx.ring-buffer.size 系统特性改变这个值。

你会看到在这个接口中直接调用 onNext()，但是考虑到回压的时候就不会这样做了。

```
Observable<Integer> myRangeWithBackpressure(int from, int count) {
    return Observable.create(new OnSubscribeRange(from, count));
}

class OnSubscribeRange implements Observable.OnSubscribe<Integer> {

    //构造器……

    @Override
    public void call(final Subscriber<? super Integer> child) {
        child.setProducer(new RangeProducer(child, start, end));
    }

}

class RangeProducer implements Producer {

    @Override
    public void request(long n) {
        //在这里调用子订阅者的onNext()
    }
}
```

你会发现，这就是 RxJava 的 range() 实现中的骨架代码。实现 Producer 是一项非常有挑战的任务：它必须是有状态的、线程安全的，而且极快。因此，一般不会自行实现 Producer，但是理解它们的运行方式是很有帮助的（参见 6.2.4 节，了解自行实现回压的细节）。在内部，回压是将 Rx 的理念颠倒过来。range()（以及其他大量内置的操作符）生成的 Observable 并不会将事件立即推送给 Subscriber。相反，只有在进行数据请求（在 Subscriber 中的 request(N) 调用）的时候，它才会苏醒过来并做出反应，生成事件。同时，它能够确保生成的事件不会超过请求数量。

接下来看一下样例是如何在 child Subscriber 上设置 Producer 的。Subscriber 调用 request() 的时候，这个 Producer 就会在 Subscriber 中被间接调用。这样就建立了一个从 Subscriber 到源 Observable 的反馈通道。Observable 指导它的 Subscriber 如何请求特定数量的数据。实际上，Observable 从推送模式变成了拉取 – 推送模式，这样客户端可以有选择地只请求数量有限的事件。如果一些外来的 Observable 没有建立这样的通道会怎么样呢？ RxJava 发现这种情况时，会将其作为不支持回压的源来进行处理，随时都可能因为 MissingBackpressureException 而失败。但是，onBackpressure*() 家族中有很多操作符都能在一定程度上模拟回压。

最简单的 onBackpressureBuffer() 操作符会无条件缓冲所有的上游事件，并将请求数量的数据传递给下游订阅者。

```
myRange(1, 1_000_000_000)
        .map(Dish::new)
        .onBackpressureBuffer()
        .observeOn(Schedulers.io())
        .subscribe(x -> {
                    System.out.println("Washing: " + x);
                    sleepMillis(50);
        });
```

和往常一样，我们依然从下往上阅读。首先，subscribe() 向上传递到 observeOn() 操作符。observeOn() 也必须要进行订阅，但是它不会简单地消费任意数量的事件。开始的时候，它只会请求固定数量的事件（128 个），避免 io() Scheduler 的队列出现溢出。onBackpressureBuffer() 操作符可以视为一个屏障，防止源忽略回压。接收到下游 Subscriber 的 request(128) 调用时，它会将这个请求往上传递；如果只有 128 个事件流过，它什么事情都不会做。但是，如果 Observable 忽略了这个请求，一味地将数据往下游推送，onBackpressureBuffer() 在内部会保持一个无边界的缓冲。下游 Subscriber 发起另外一个请求时，onBackpressureBuffer() 首先会消耗其内部缓冲的数据，只有缓冲几乎消耗殆尽的时候，它才会向上游发起请求。这种聪明的机制能够让 observeOn() 运行得就像 myRange() 支持回压一样，但实际上节流是由 onBackpressureBuffer() 实现的。令人遗憾的是，对无限的内部缓冲并不能掉以轻心。

```
Created: 1
Created: 2
Created: 3
Created: 4
Created: 8
Created: 9
Washing: 1
Created: 10
Created: 11
...
Created: 26976
Created: 26977
Washing: 15
Exception in thread "main" java.lang.OutOfMemoryError: ...
Washing: 16
    at java.util.concurrent.ConcurrentLinkedQueue.offer...
    at rx.internal.operators.OperatorOnBackpressureBuffer...
...
```

当然，你的情况可能有所不同。如果事件的数量更少并且有足够的内存，那么在技术上 onBackpressureBuffer() 是可以运行的。但我们永远都不应该依赖没有限制的资源。固态硬盘的容量是有限的，不过好在还有一个重载版本的 onBackpressureBuffer(N)，它能够指定最大的缓冲空间。

```
.onBackpressureBuffer(1000, () -> log.warn("Buffer full"))
```

第二个参数是可选的，它是一个回调。当有界的 1000 个元素的缓冲被填满时，它会被调用，尽管 Subscriber 的处理速度依然很慢。它不会尝试进行任何的恢复操作，所以在警告信息之后会立即出现 MissingBackpressureException。通过这种方式，至少对缓冲有了自己的控制，不再受硬件或操作系统的控制。

onBackpressureBuffer() 的一个替代方案是 onBackpressureDrop()，后者会将没有预先请求（request()）的事件直接丢弃。假设在餐厅中，服务员将新的待清洗的盘子不断送到厨房。onBackpressureBuffer() 就像一张能够有限或无限堆放的桌子，待清洗的盘子都会放到这里。onBackpressureDrop() 则与之不同，如果此时没有清洗能力，服务员会直接把脏盘子扔掉。这不是可持续的商业模式，但是至少餐厅能够一直为顾客服务。

```
.onBackpressureDrop(dish -> log.warn("Throw away {}", dish))
```

回调是可选的，它会在每次有事件丢弃的时候进行通知，丢弃是因为没有请求它就出现了。跟踪丢弃事件的数量是一种很好的做法，这是一项重要的指标。最后，还有 onBackpressureLatest() 方法，它与 onBackpressureDrop() 非常类似，但是它会引用最后一个被丢弃的元素，这样，如果稍后调用下游的 request()，就能使用上游的最后一个值。

onBackpressure*() 方法族提供了一个桥梁，将要求回压功能的操作符和订阅者与不支持回压的 Observable 连接在一起。但是，最好还是使用和创建原生支持回压的源。

6.2.4　按照请求返回指定数量的数据

构造支持下游回压请求的 Observable，方式有很多。最简单的方案是使用内置的工厂方法，如 range() 或 from(Iterable<T>)。后者会创建一个由 Iterable 作为支撑的源，但是内置了回压的功能。这意味着该 Observable 并不会将 Iterable 中的所有值一次性都发布出去，相反，它会在消费者请求的时候将值逐渐发布。注意，这并不意味着会预先将所有的数据都加载到 List<T>（扩展了 Iterable<T>）中。一般来讲，Iterable 是 Iterator 的一个工厂，所以我们可以在运行时安全地加载数据。

在实现支持回压的 Observable 方面，有一个很有趣的例子，那就是将来自 JDBC 的 ResultSet 包装到一个流。注意，ResultSet 是基于拉取模型的，这与支持回压的 Observable 非常类似。但它并不是 Iterable 或 Iterator，所以必须先转换成 Iterator<Object[]>，Object[] 是数据库中单个行的松散类型化表述。

```
public class ResultSetIterator implements Iterator<Object[]> {

    private final ResultSet rs;

    public ResultSetIterator(ResultSet rs) {
        this.rs = rs;
    }

    @Override
    public boolean hasNext() {
        return !rs.isLast();
    }

    @Override
    public Object[] next() {
        rs.next();
        return toArray(rs);
    }
}
```

上述的转换器是一个非常简单的版本，没有错误处理，它从 Apache Commons DbUtils 开源工具库中的 ResultSetIterator 抽取而来。这个类还提供了对 Iterable<Object[]> 的简单转换。

```
public static Iterable<Object[]> iterable(final ResultSet rs) {
    return new Iterable<Object[]>() {

            @Override
```

```
            public Iterator<Object[]> iterator() {
                return new ResultSetIterator(rs);
            }

        };
    }
```

ResultSet 的处理

需要记住，将 ResultSet 视为 Iterator（尤其是 Iterable）是一个有漏洞的抽象。首先，ResultSet 像 Iterator 一样具有破坏性，但是与 Iterable 不同。你只能遍历 Iterator 一次，ResultSet 通常也是如此。其次，Iterable 是全新 Iterator 的工厂，而前面的转换器始终会返回由同一个 ResultSet 支撑的 Iterator。这意味着，调用 iterator() 两次不会生成相同的值，两个迭代器会在同一个 ResultSet 上竞争。最后，ResultSet 在完成的时候必须要关闭，但是 Iterator 没有这样的生命周期。完全依赖客户端代码读取 Iterator 来进行清理就显得过于乐观了。

这些转换器就绪之后，就可以基于 ResultSet 构建支持回压功能的 Observable<Object[]> 了。

```
Connection connection = //...
PreparedStatement statement =
        connection.prepareStatement("SELECT ...");
statement.setFetchSize(1000);
ResultSet rs = statement.executeQuery();
Observable<Object[]> result =
    Observable
        .from(ResultSetIterator.iterable(rs))
        .doAfterTerminate(() -> {
            try {
                rs.close();
                statement.close();
                connection.close();
            } catch (SQLException e) {
                log.warn("Unable to close", e);
            }
        });
```

这里得到的 result Observable 就支持回压了，因为内置的 from() 操作符支持回压。这样，Subscriber 的吞吐量就无关紧要了，也不会再看到 MissingBackpressureException。注意，setFetchSize() 是非常必要的，因为有些 JDBC 驱动可能会尝试将所有记录都加载到内存，如果想要对很大的结果集进行流处理，这是非常低效的。

正如前面介绍的，支持回压的底层机制是自定义实现的 Producer。但是这项任务非常容易出错，因此 RxJava 创建了一个辅助类，即 SyncOnSubscribe。Observable.OnSubscribe 的这个实现是基于拉取模式的，并且透明地将回压加入进来。从最简单的无状态 Observable 开始介绍——在现实生活中非常少见。这种类型的 Observable 在每次 onNext() 调用之间不会持有任何状态。但是，即便是最简单的 range() 或 just() 也必须要记得发布过哪些条目。无状态 Observable 为数不多的有用场景之一就是发布随机数。

```
import rx.observables.SyncOnSubscribe;


Observable.OnSubscribe<Double> onSubscribe =
    SyncOnSubscribe.createStateless(
        observer -> observer.onNext(Math.random())
    );

Observable<Double> rand = Observable.create(onSubscribe);
```

rand Observable 是一个普通的 Observable，它可以进行转换、组合和订阅。但是，在内部，它对回压提供了完整的支持。如果 Subscriber 或管道上的其他操作符请求有限数量的事件，这个 Observable 就会完全遵守这个要求。只需要给 createStateless() 提供一个 lambda 表达式：对于请求的每个事件，它都会进行调用。如果下游调用 request(3)，这个自定义的表达式就会被调用三次，这里假设每次只发布一个事件。在每次调用之间没有上下文（状态），因此它被称为无状态的。

现在，构建一个有状态的操作符。SyncOnSubscribe 的这个变种允许在各种调用间传递一个不可变的状态变量。同时，每次调用必须要返回一个新的状态值。样例会构建一个从零开始的无限自然数生成器。如果你想使用单调递增的自然数来压缩一个任意长的序列，这样的操作符是非常有用的。range() 也能很好地运行，但是它需要提供一个上限，在有些情况下，这可能不太实用，如下所示。

```
Observable.OnSubscribe<Long> onSubscribe =
        SyncOnSubscribe.createStateful(
                () -> 0L,
                (cur, observer) -> {
                    observer.onNext(cur);
                    return cur + 1;
                }
        );

Observable<Long> naturals = Observable.create(onSubscribe);
```

这里为 createStateful() 工厂方法提供了两个 lambda 表达式。第一个表达式以延迟执行的方式创建了一个初始状态，在本例中这个状态就是零。第二个表达式更加重要：它基于当前的状态，按照某种方式推送一个条目到下游，并返回一个新的状态值。这个状态应该是不可变的，所以这个方法允许返回一个新的状态，而不是改变当前的状态。你可以很容易地重写 naturals Observable，让它返回 BigInteger，避免可能会出现的溢出。这个 Observable 可以生成任意数量的递增自然数，但是它完全支持回压。这意味着它可以基于 Subscriber 的需求，调整事件的生成速度。与原始的实现方式相比，确实会简单得多，但是遇到缓慢的 Subcriber 时，它就暴露出不足了。

```
Observable<Long> naturals = Observable.create(subscriber -> {
    long cur = 0;
    while (!subscriber.isUnsubscribed()) {
        System.out.println("Produced: " + cur);
        subscriber.onNext(cur++);
    }
});
```

如果你希望使用单个状态变量，这个变量在遍历的时候能够进行改变（如 JDBC 中的 ResultSet），SyncOnSubscribe 同样也为你提供了一个方法。如下的代码由于检查型异常**无法**编译通过，但是这里只想强调整体的使用模式。

```
ResultSet resultSet = //...

Observable.OnSubscribe<Object[]> onSubscribe = SyncOnSubscribe.createSingleState(
        () -> resultSet,
        (rs, observer) -> {
            if (rs.next()) {
                observer.onNext(toArray(rs));
            } else {
                observer.onCompleted();
            }
            observer.onNext(toArray(rs));
        },
        ResultSet::close
);

Observable<Object[]> records = Observable.create(onSubscribe);
```

这里有三个回调需要实现。

- 状态的生成器。这个 lambda 表达式只会被调用一次，用于生成状态变量，这个变量将会作为参数传递给后续的表达式。
- 生成下一个值的回调，此时通常会基于当前状态来生成下一个值。这个回调可以自由地改变第一个参数给定的状态。
- 第三个回调会在取消订阅时调用，这里用来清理 ResultSet。

具备错误处理功能的更加完整的实现如下所示。注意，在取消订阅时发生的错误很难恰当地传递到下游中。

```
Observable.OnSubscribe<Object[]> onSubscribe = SyncOnSubscribe.createSingleState(
    () -> resultSet,
    (rs, observer) -> {
        try {
            rs.next();
            observer.onNext(toArray(rs));
        } catch (SQLException e) {
            observer.onError(e);
        }
    },
    rs -> {
        try {
            //同时要关闭Statement、Connection等
            rs.close();
        } catch (SQLException e) {
            log.warn("Unable to close", e);
        }
    }
);
```

SyncOnSubscribe 是一个非常便利的工具集，它允许编写支持回压的 Observable。[4] 相对于 Observable.create()，它会更复杂一些，但是每个 Subscriber 控制回压带来的收益也不能低估。避免直接使用 create() 操作符，我们应该考虑使用内置的工厂方法，如 from() 或 SyncOnSubscribe。

回压能够通过 Subscriber 对 Observable 进行节流，这是一个非常强大的机制。这样的反馈通道显然会带来一定的开销，但是既能保证松耦合又能管理生产者和消费者，这有着巨大的优势。回压通常伴随着批处理，所以额外的开销已经最小了。但是如果 Subscriber 非常缓慢（甚至短暂），这种缓慢会立即体现出来，整个系统的稳定性都会受到牵连。使用 onBackpressure*() 家族方法可以在一定程度上迁移不支持回压的 Observable，但这并不是长久之计。

创建自己的 Observable 时，要正确地处理回压请求。毕竟我们无法控制 Subscriber 的吞吐量。另外一项技术是避免在 Subscriber 中执行重量级的任务，而应该将其放到 flatMap() 中。例如，与其在 subscribe() 中进行数据库存储的操作，我们不如这样做，如下所示。

```
source.subscribe(this::store);
```

应该考虑让 store 更加符合反应式的要求（让它返回已保存记录的 Observable<UUID>），并且只订阅触发订阅和副作用。

```
source
    .flatMap(this::store)
    .subscribe(uuid -> log.debug("Stored: {}", uuid));
```

或者更进一步，批量获取 UUID 以减少日志框架的开销。

```
source
    .flatMap(this::store)
    .buffer(100)
    .subscribe(
        hundredUuids -> log.debug("Stored: {}", hundredUuids))
```

避免在 subscribe() 中执行长时间运行的任务，能够减少回压的需求，但是预先考虑回压依然是个很好的做法。请参考 JavaDoc 以了解操作符是否支持回压，如果操作符缺少这样的信息，很可能就不会受到回压的影响，比如 map()。

6.3 小结

本章的一个重要结论就是要避免使用 Observable.create() 和手动发布事件。如果必须自己实现 Observable，请考虑使用支持回压功能的工厂方法。同时，要关注你的领域，也许可以安全地跳过事件或对事件进行批处理，以减少消费端的整体负载。

注 4：如果你需要一个更具反应式的工具集，请核查 AsyncOnSubscribe，原则上它非常相似，但是为 Observer 生成下一个条目的回调也可以是异步的。

第 7 章

测试和排错

托马什·努尔凯维茨（Tomasz Nurkiewicz）

本书读到现在，相信你已经理解了使用 Reactive Extensions 进行编程的基本原则。到目前为止，本书已经介绍了订阅、常用的操作符、在已有的应用程序中使用 RxJava，以及如何编写完整的反应式软件栈。但是，为了发挥反应式编程的最大威力，我们必须更进一步。本章将会关注一些不常见但是同样重要的方面和原则，如下所示。

- 声明式的错误处理，包括重试（参见 7.1 节）。
- 虚拟时间和测试（参见 7.2.1 节）。
- 监控和调试 `Observable` 流（参见 7.4 节）。

如果要将一个库或框架成功部署到生产环境中，仅仅理解它是不够的。如果你想构建可靠、稳定且有弹性的应用程序，上述各个方面都至关重要。

7.1　错误处理

反应式宣言列举了反应式系统应该具备的 4 个特征：即时响应性（responsive）、回弹性（resilient）、弹性（elastic）和消息驱动（message driven）。看一下其中两个特征。

- ❑ 即时响应性

  只要有可能，系统就会及时地做出响应。[…] 即时响应意味着可以快速检测到问题，并且进行有效地处理。[…] 快速而一致的响应时间，[…] 简化错误处理。

- ❑ 回弹性

  系统在出现失败时依然要保持即时响应性。[…] 系统某部分的失败不会危及整个系统，并能独立恢复。[…] 组件的客户端不再承担组件失败处理的任务。

本节将会阐述即时响应性和回弹性为何如此重要，以及 RxJava 是如何帮助实现它们的。现在，你已经非常熟悉在订阅 Observable 的时候使用 onError() 回调。但这只是冰山一角，而且通常不是处理错误的最佳方式。

7.1.1 我的异常在哪里

在传统的 Java 中，错误通常用异常来表示。Java 中有以下两种类型的异常。

- 非检查型异常，这种异常在方法声明上不需要体现出来。如果某个方法抛出非检查型异常（比如 NullPointerException），可以在方法声明上标记该异常，但这并不是强制的。
- 检查型异常，为了保证代码通过编译，这种异常必须要声明和处理。基本上，这是所有未扩展 RuntimeException 或 Error 的 Throwable，比如 IOException。

这两种传统异常处理方式各有利弊。非检查型异常很容易添加进来，并且不会破坏编译时的向后兼容性。同时，使用非检查型异常时，客户端代码看上去会更整洁，因为它不需要进行错误处理（尽管它可以这样做）。而检查型异常更明确地说明了方法预期的输出。当然，每个方法可以抛出任意类型，但是检查型异常被视为 API 的一部分，明确指出了必须处理的错误。尽管检查型异常无法被忽略，而且在编写无错误的代码方面似乎更具优势，但是事实证明，它们非常笨拙和晦涩。即便是官方的 Java API 也正在迁移至非检查型异常。例如，旧的 JMSException（检查型）在 JMS 2.0 中变成了新的 JMSRuntimeException（非检查型）。

RxJava 采取了一种完全不同的方式。首先，在标准的 Java 中，异常是类型系统中的一个新维度。方法有一个返回类型，同时还有与之完全正交的异常。某个方法打开一个 File 时，它可能会返回 InputStream，也可能会抛出 FileNotFoundException。但是，FileNotFoundException 没有声明会怎么样呢？或者说，是否还要预期出现其他异常？异常就像另外一条执行路径，正如失败始终是意料之外的事情一样，异常也永远不会是正常业务流的一部分。在 RxJava 中，失败是另外一种类型的通知。Observable<T> 都是 T 类型的事件序列，后面跟随可选的完成或失败通知。这意味着错误隐式地变成了每个流的一部分，即便不需要对其进行处理，很多操作符也能够以更加稳定的方式声明式地处理错误。同时，在 Observable 周围贸然添加 try-catch 并不会捕获到任何错误，这些错误只能通过上述的错误通知进行传递。

但是，在探讨如何通过 RxJava 操作符声明式地处理错误之前，我们必须要理解错误在完全不进行处理时使用启发式的行为。在 Java 中，异常可能会在任意地方出现。库的作者必须确保错误得到了恰当的处理；如果不进行处理，至少也要进行报告。使用 subscribe() 最常见的问题就是没有定义 onError 回调。

```
Observable
    .create(subscriber -> {
        try {
            subscriber.onNext(1 / 0);
        } catch (Exception e) {
            subscriber.onError(e);
        }
    })
```

```
    //有问题的，缺少onError()回调
    .subscribe(System.out::println);
```

在 create() 中，样例强制抛出 ArithmeticException 并调用每个 Subscriber 的 onError() 回调。但是，令人遗憾的是，subscribe() 根本没有提供 onError() 实现。不过，RxJava 会努力扭转败局，它会抛出包装原始 ArithmeticException 的 OnErrorNotImplementedException。但是，由哪个线程抛出这个异常呢？这个问题很难回答。如果 Observable 是同步的（如前面的样例所示），那么客户端线程将会间接调用 create()，因此会在没有处理 onError() 的时候抛出 OnErrorNotImplementedException。这意味着调用 subscribe() 的线程将接收到 OnErrorNotImplementedException。

如果你忘记订阅错误而且 Observable 是异步的，那情况就变得更加复杂了。这时抛出 OnErrorNotImplementedException，调用 subscribe() 的线程可能早就消失了。在这种情况下，异常会在准备调用 onError() 回调的线程中抛出。这可能是通过 subscribeOn() 或最后一个 observeOn() 选择的来自 Scheduler 的线程。Scheduler 可以按照任意方式管理这种预料之外的异常，大多数情况下，它只是将堆栈跟踪打印到标准错误流中。这远远称不上完美的方案：这样的异常绕过了正常日志代码，甚至会被忽略。因此，subscribe() 只监听值而忽略错误通常是一个不好的信号，可能会丢失错误。即便预期不会发生任何异常（这种情况非常罕见），至少也应该将错误日志插入日志框架。

```
private static final Logger log = LoggerFactory.getLogger(My.class);

//...

.subscribe(
        System.out::println,
        throwable -> log.error("That escalated quickly", throwable));
```

还有很多其他的地方可能会产生异常或者不经意间将异常引入进来。首先，在 create() 中使用 try-catch() 代码块将 lambda 表达式包装起来，通常是一种好的实践，比如在上面的样例中使用以下代码。

```
Observable.create(subscriber -> {
    try {
        subscriber.onNext(1 / 0);
    } catch (Exception e) {
        subscriber.onError(e);
    }
});
```

但是，如果你忘记使用 try-catch 并让 create() 抛出异常，RxJava 会竭尽所能地将异常以 onError() 通知的形式进行传递。

```
Observable.create(subscriber -> subscriber.onNext(1 / 0));
```

上面的两个代码样例在语义上是等价的。create() 抛出的异常在内部会被 RxJava 捕获并转换成错误通知。不过，还是建议尽可能通过 subscriber.onError() 显式地传递异常。更好的方法是使用 fromCallable()。

```
Observable.fromCallable(() -> 1 / 0);
```

其他可能生成异常的是接收用户代码的各种操作符。简而言之，就是接收 lambda 表达式作为参数的各种操作符，比如 map()、filter()、zip() 等。这些操作符不仅要处理来自上游 Observable 的错误通知，还要处理自定义映射函数或断言抛出的异常。以下面这个有问题的映射和过滤为例。

```
Observable
        .just(1, 0)
        .map(x -> 10 / x);

Observable
        .just("Lorem", null, "ipsum")
        .filter(String::isEmpty);
```

对于某些元素，第一个例子抛出了 ArithmeticException。在第二个例子中，调用 filter() 断言时，会导致 NullPointerException。传递给高阶函数（如 map() 和 filter()）的 lambda 表达式都应该是纯表达式，而抛出异常是一种不纯粹的副作用。在这里，RxJava 会再次竭力处理预料之外的异常，而且它的行为符合预期。如果管道中的任意操作符抛出异常，它会被转换成错误通知并传递到下游中。尽管 RxJava 试图去修复有问题的用户代码，如果 lambda 表达式有抛出异常的可能，那么我们应该使用 flatMap() 进行显式声明。

```
Observable
        .just(1, 0)
        .flatMap(x -> (x == 0) ?
                Observable.error(new ArithmeticException("Zero :-(")) :
                Observable.just(10 / x)
        );
```

flatMap() 操作符用途非常广泛，它不需要指明异步计算的下一步要做什么。Observable 是一个包含值或错误的容器，所以如果你想要声明式地表达即便非常快速的计算也有可能产生错误，那么使用 Observable 包装也是一个不错的可选方案。

7.1.2　替代声明式的try-catch

错误与流经 Observable 管道的正常事件非常相似。理解了错误来自何方，接下来就要学习如何声明式地对其进行处理。我们使用的 Observable 通常是多个操作符和上游 Observable 的组合体。以下样例是基于某些数据构建保险协议的过程。

```
Observable<Person> person = //...
Observable<InsuranceContract> insurance = //...
Observable<Health> health = person.flatMap(this::checkHealth);
Observable<Income> income = person.flatMap(this::determineIncome);
Observable<Score> score = Observable
    .zip(health, income, (h, i) -> asses(h, i))
    .map(this::translate);
Observable<Agreement> agreement = Observable.zip(
    insurance,
    score.filter(Score::isHigh),
    this::prepare);
Observable<TrackingId> mail = agreement
    .filter(Agreement::postalMailRequired)
```

```
        .flatMap(this::print)
        .flatMap(printHouse::deliver);
```

这个略显牵强的样例展现了业务处理的几个步骤。首先，样例加载 Person，查找可用的 InsuranceContract，基于 Person 确认 Health 和 Income（并发分叉执行）。然后，样例将这两个结果联合起来进行计算并转换成 Score。最后，InsuranceContract 与 Score（仅在它的值比较高的情况下）联合起来，并执行一些后置的处理，比如发送邮件。你应该知道，到此为止，其实什么都没有开始执行。样例只是定义了要进行的操作，在真正有人订阅之前，不会执行任何业务逻辑。但是，如果某一个上游源出现了错误通知又会怎样呢？这里没有可见的错误处理，但是错误能够非常便利地进行传递。

目前见过的所有操作符主要是作用于值的，完全忽略了错误。这样也是可以的：普通的操作符将流经的值进行转换，但是会跳过完成和错误通知，让它们往下游流动。这意味着任意上游 Observable 的单个错误会与级联失败一起传递给所有下游的订阅者。再次强调，如果业务逻辑需要所有的步骤都成功，这种方式是没有问题的。但是，在有些情况下，你可以安心地忽略失败，并将其替换为备用值或次级源。

1. 使用onErrorReturn()将错误替换为固定的结果

在 RxJava 中，最简单的错误处理操作符是 onErrorReturn()：遇到错误的时候，会使用一个固定的值来进行替换。

```
Observable<Income> income = person
        .flatMap(this::determineIncome)
        .onErrorReturn(error -> Income.no())

//...

private Observable<Income> determineIncome(Person person) {
    return Observable.error(new RuntimeException("Foo"));
}

class Income {
    static Income no() {
        return new Income(0);
    }
}
```

onErrorReturn() 操作符甚至都不需要解释。正常的事件通过时，这个操作符不会做任何事情。但是，接收到来自上游的错误通知时，它会立即丢弃错误并替换为一个固定值——在本例中是 Income.no()。相对于按照命令式的风格在 try-catch 代码块的 catch 语句中返回一个固定值，onErrorReturn() 是一个非常流畅且易读的替代方案。

```
try {
    return determineIncome(Person person)
} catch(Exception e) {
    return Income.no();
}
```

在本例中，catch 语句吞噬了原始的异常，并且返回了一个固定值。可能它就是这样设计的，但通常比较好的做法是在异常发生时至少进行日志记录。RxJava 中的所有错误处

理操作符的都是这样表现的：如果已经显式处理某些异常，那么该异常会被吞噬。这是你确实需要考虑的，如果故障系统的日志文件没有暴露任何的问题，那可以说是最糟糕的事情了。onErrorReturn() 将错误作为参数传递了过来，但是却被忽略了。你可以在 onErrorReturn() 中记录异常，也可以使用更加专业的诊断操作符，7.4 节将对此进行介绍。现在，只需要记住，RxJava 中的所有操作符都将异常记录和监控的任务留给了我们。

2. 使用onErrorResumeNext()延迟计算备用值

使用 onErrorReturn() 返回一个固定值有时是一种很好的方式，但通常我们想要在出现错误时延迟计算出备用值。以下是两种可能的场景。

- 生成数据流的主要方式失败了（发生了 onError() 事件），所以切换到同样非常好的备用源上，但是因为某种原因，我们将其视为备用方案（比如该方案更慢、代价更高等）。
- 出现失败的时候，我们想要将真实的数据替换为代价更低、稳定性更好的，但可能有些陈旧的信息。比如，检索新数据失败的时候，可能会从缓存中选择较为陈旧的流。另一个常见样例可能会带来稍差的用户体验。例如，在线商城中会返回全局最畅销的商品，而不是个性化推荐的商品。

显然，错误发生时需要的逻辑本身可能代价非常高昂，而且可能会出错。因此，必须将备用逻辑封装到一个延迟执行的包装器中，并且最好是异步的。这样的包装器会是什么样？当然是 Observable！

```
Observable<Person> person = //...
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .onErrorResumeNext(person.flatMap(this::guessIncome));

//...

private Observable<Income> guessIncome(Person person) {
        //...
}
```

onErrorResumeNext() 操作符基本上就是将错误通知替换成了另外一个流。如果你订阅了某个使用 onErrorResumeNext() 作为防护措施的 Observable，RxJava 会透明地从主 Observable 切换到备用 Observable（指定为参数）。在样例中，如果 income 流失败，错误通知会被捕获，库会自动订阅 guessIncome() 流。这个流可能不那么精确，但是更加可靠、迅速或低成本。有意思的是，onErrorResumeNext() 可以替换成 concatWith()，假设 determineIncome 会且仅会发布一个值或错误。

```
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .flatMap(
        Observable::just,
        th -> Observable.empty(),
        Observable::empty)
    .concatWith(person.flatMap(this::guessIncome))
    .first();
```

flatMap() 操作符有个不常见的地方：它接收三个 lambda 表达式而不是一个。

- 第一个参数允许将上游 Observable 中的每个元素替换为新的 Observable，这是本书中 flatMap() 一直以来的用法。
- 第二个参数将可能会出现的错误通知替换为另外一个流。我们想要忽略上游的错误，所以在这里只切换为一个空的 Observable。
- 最后，上游正常完成时，完成通知可以替换为另外一个流。

first() 操作符在这里的使用至关重要。使用 first() 操作符表明只想获取第一个出现的事件。在成功的情况下，我们会得到 determineIncome 的结果，而 RxJava 永远不会订阅 guessIncome() 的结果。但是出现失败的时候，第一个 Observable 实际上没有发布任何事件，所以 first() 会请求其他条目，此时就会使用订阅的备用流，这个流作为参数传递给 concatWith()。

到这里，相信你已经意识到本例中的 concatWith() 并不是真正必要的，最复杂形式的 flatMap() 就足够了。即便 first() 也并非必须要有。思考如下的代码。

```
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .flatMap(
        Observable::just,
        th -> person.flatMap(this::guessIncome),
        Observable::empty);
```

上述的样例有一个非常有意思的特性：基于 Throwable 类型的 th，可以返回 onError() 映射中一个不同的 Observable。所以理论上能够基于异常信息或类型，返回不同的备用流。onErrorResumeNext() 操作符有一个重载版本，能够实现这样的功能。

```
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .onErrorResumeNext(th -> {
        if (th instanceof NullPointerException) {
            return Observable.error(th);
        } else {
            return person.flatMap(this::guessIncome);
        }
    });
```

尽管 flatMap() 的用途非常广泛，能够提供灵活的错误处理功能；但是 onErrorResumeNext() 的表述性更好，代码更易于阅读。所以我们应该优先使用后者。

7.1.3 事件没有发生导致的超时

RxJava 提供了一些操作符来处理上游 Observable 的异常通知。但是，你知道比错误更糟糕的情况是什么吗？静默！要连接的系统因为出现了异常而失败，这相对来说更容易进行预测、处理、单元测试等。但是，如果你订阅了一个 Observable，本来预期能够立即得到结果，但是它却迟迟不发布任何内容，那么你该怎么办？这种场景比简单地出现错误糟糕得多。系统的延迟会受到严重影响，系统似乎被挂起了，但是日志中没有任何清晰的标记。

幸而，RxJava 提供了一个内置的 timeout() 操作符监听上游的 Observable，它会持续监控自上一个事件发布或 Observable 订阅以来，经历了多长时间。如果连续事件之间的静默

时间超出了给定的时间段，timeout() 操作符会发布一个包含 TimeoutException 的错误通知。为了更好地理解 timeout() 是如何运行的，首先考虑在特定时间之后只发布一个事件的 Observable。为了方便阐述，我们将会创建一个 Observable，它会在 200 毫秒之后返回 Confirmation 事件。通过添加 delay(100, MILLISECONDS) 模拟延迟。除此之外，我们还想要在事件和完成通知之间模拟额外的延迟。这正是使用 empty() Observable 的目的所在，正常情况下，它会立即完成；但是因为额外的 delay()，发送完成通知之前会进行等待。将这两个流组合起来之后，效果如下。

```
Observable<Confirmation> confirmation() {
    Observable<Confirmation> delayBeforeCompletion =
        Observable
            .<Confirmation>empty()
            .delay(200, MILLISECONDS);
    return Observable
            .just(new Confirmation())
            .delay(100, MILLISECONDS)
            .concatWith(delayBeforeCompletion);
}
```

现在，测试一下按照最简单的重载版本驱动 timeout() 操作符。

```
import java.util.concurrent.TimeoutException;

//...

confirmation()
    .timeout(210, MILLISECONDS)
    .forEach(
        System.out::println,
        th -> {
            if ((th instanceof TimeoutException)) {
                System.out.println("Too long");
            } else {
                th.printStackTrace();
            }
        }
    );
```

这里使用 210 毫秒的超时并不是巧合。从开始订阅到 Confirmation 实例到达的时间间隔恰好是 100 毫秒，小于超时的阈值。另外，这个事件和完成通知之间的延迟是 200 毫秒，依然小于 210。因此，在本例中，timeout() 操作符是透明的，并不会影响整个消息流。但是，如果 timeout() 的阈值调整为略小于 200 毫秒（假设为 190 毫秒），那么它的作用就体现出来了。Confirmation 展现了出来，但是完成回调没有执行，我们接收到了一个包含 TimeoutException 的错误通知。第一个事件抵达的延迟远小于 200 毫秒，而第一个事件和第二个事件（实际上是完成通知）之间的延迟超过了 190 毫秒，因此会将一个错误通知传递到下游中。当然，如果把超时阈值设置为小于 100 毫秒，我们甚至无法看到第一个事件。

这是 timeout() 最简单的用例：当我们想要限制等待响应的时间时，就会发现它非常有用。但有的时候，固定的超时阈值过于严格，而我们可能想要在运行时调整超时时间。假设想要构建一个预测下次日食的算法。算法的接口是一个 Observable<LocalDate>（理

应如此），它代表了此类事件未来的日期。假设这个算法是计算密集类型的，我们依然会进行模拟，不过这次使用的是 interval() 操作符（参见 2.4.3 节），将一个固定的日期列表和 interval() 生成的一个较缓慢的进度流合并到一起。借助 interval(500, 50, MILLISECONDS)，第一个日期会在 500 毫秒之后出现，后续的每个日期会在 50 毫秒之后出现。在现实系统中，这非常常见：响应的第一个元素会有比较高的延迟。这可能是初始化连接、SSL 握手、查询优化或服务器端要执行的事情导致的。但是，后续的事件要么能够轻而易举获得，要么很容易检索，所以它们之间的延迟要低得多。

```
Observable<LocalDate> nextSolarEclipse(LocalDate after) {
    return Observable
        .just(
            LocalDate.of(2016, MARCH, 9),
            LocalDate.of(2016, SEPTEMBER, 1),
            LocalDate.of(2017, FEBRUARY, 26),
            LocalDate.of(2017, AUGUST, 21),
            LocalDate.of(2018, FEBRUARY, 15),
            LocalDate.of(2018, JULY, 13),
            LocalDate.of(2018, AUGUST, 11),
            LocalDate.of(2019, JANUARY, 6),
            LocalDate.of(2019, JULY, 2),
            LocalDate.of(2019, DECEMBER, 26))
        .skipWhile(date -> !date.isAfter(after))
        .zipWith(
            Observable.interval(500, 50, MILLISECONDS),
            (date, x) -> date);
}
```

在这种类型的场景下，如果使用固定的阈值就是有问题的做法了。第一个事件应该使用较为保守的限制值，而后续事件的间隔应该更加乐观一些。重载版本的 timeout() 恰好能够实现这一点：它会接收两个 Observable 的工厂，一个会生成第一个事件的超时时间，另一个则会生成后续元素的超时时间。代码胜千言，如下所示。

```
nextSolarEclipse(LocalDate.of(2016, SEPTEMBER, 1))
    .timeout(
        () -> Observable.timer(1000, TimeUnit.MILLISECONDS),
        date -> Observable.timer(100, MILLISECONDS))
```

在这里，第一个 Observable 会在 1 秒之后发布一个事件，它代表了第一个事件能接受的延迟阈值。第二个 Observable 是为流中出现的每个元素创建的，允许细粒度地控制后续事件的超时。注意，这里并没有使用 date 参数。可以设想一下在某种场景中具备自适应能力的超时时间，比如，如果上一个延迟比以往更长，那么等待下一个事件时，可以稍微多等待一些时间。或者，与之相反，让后续事件的超时时间更短一些，以适应订阅者的性能。

在有些情况下，即便没有超时，跟踪每个事件的延迟也是非常有用的。此时，可以使用非常便利的 timeInterval() 操作符：它会将 T 类型的事件替换成 TimeInterval<T>。后者会封装事件，但是也会显示从上一个事件开始经过了多少时间（对于第一个事件，是从订阅开始经历的时间）。

```
Observable<TimeInterval<LocalDate>> intervals =
        nextSolarEclipse(LocalDate.of(2016, JANUARY, 1))
                .timeInterval();
```

除了能够返回 LocalDate 实例中的 getValue() 方法之外，TimeInterval<LocalDate> 还有一个 getIntervalInMilliseconds()。不过，研究一下上述程序在订阅之后的输出，能够容易地看出它是如何运行的。你可以清晰地看到，第一个事件耗费了 533 毫秒才出现，而后续的每个事件基本只需要约 50 毫秒。

```
TimeInterval [intervalInMilliseconds=533, value=2016-03-09]
TimeInterval [intervalInMilliseconds=49, value=2016-09-01]
TimeInterval [intervalInMilliseconds=50, value=2017-02-26]
TimeInterval [intervalInMilliseconds=50, value=2017-08-21]
TimeInterval [intervalInMilliseconds=50, value=2018-02-15]
TimeInterval [intervalInMilliseconds=50, value=2018-07-13]
TimeInterval [intervalInMilliseconds=50, value=2018-08-11]
TimeInterval [intervalInMilliseconds=50, value=2019-01-06]
TimeInterval [intervalInMilliseconds=51, value=2019-07-02]
TimeInterval [intervalInMilliseconds=49, value=2019-12-26]
```

timeout() 还有一个重载版本，它能在遇到错误时接收一个备用的 Observable 以替代初始的源。这种行为与 onErrorResumeNext() 非常类似（参见 7.4.2 节）。

7.1.4 失败之后的重试

onError 是一个终结通知，意味着在这个流中不会出现其他事件了。因此，如果你想要标记潜在非致命的业务条件，那么应该避免使用 onError。这与通常建议避免使用异常来控制程序流并没有太大的差异。相反，在 Observable 中，可以考虑将错误包装到特殊类型的事件中，这样的事件能够随着正常事件多次出现。例如，如果你想要提供一个由交易结果组成的流，但是有些交易可能会因为业务原因（比如余额不足）而失败。这种情况下，应该避免使用 onError 通知。相反，应该考虑创建一个 TransactionResult 抽象类，这个抽象类可以有两个具体的子类，分别代表成功和失败。在这种流中，onError 通知表示出现了非常严重的问题，比如阻碍后续事件发布的灾难性故障。

也就是说，onError 可以代表外部组件或系统的瞬时故障。有意思的是，通常简单重试一次就能成功了。其他的系统可能正在经历短暂的负载高峰、GC 暂停或重启。在构建健壮且有弹性的应用程序时，重试是一种很重要的机制。RxJava 对重试提供了一流的支持。

最简单版本的 retry() 操作符会重新订阅失败的 Onservable，并希望它能继续生成正常的事件，而不是再次出现失败。出于教学讲解的目的，以下创建一个存在严重问题的 Observable。

```
Observable<String> risky() {
    return Observable.fromCallable(() -> {
        if (Math.random() < 0.1) {
            Thread.sleep((long) (Math.random() * 2000));
            return "OK";
        } else {
            throw new RuntimeException("Transient");
        }
    });
}
```

在百分之九十的情况下，订阅 risky() 都会导致 RuntimeException。如果进入 OK 分支，那么样例将会注入零到两秒的人为延迟。以下使用这个有风险的操作来阐述 retry()。

```
risky()
    .timeout(1, SECONDS)
    .doOnError(th -> log.warn("Will retry", th))
    .retry()
    .subscribe(log::info);
```

需要记住，缓慢的系统其实和有问题的系统没什么区别，但是前者往往更糟糕，因为我们将会经历额外的延迟。在拥有超时功能之后，有时甚至需要带有重试机制的主动超时——当然，重试不能有副作用而且操作是幂等的。retry() 的行为方式非常简单：它会将所有的事件和完成通知推送至下游，但是不包含 onError()。错误通知被吞噬了（所以也不会有异常记录），因此我们需要使用 doOnError() 回调（参见 7.4.1 节）。每当 retry() 遇到模拟的 RuntimeException 或 TimeoutException 时，它就会尝试重新订阅。

警告一下：如果 Observable 进行了缓存，或者以某种方式保证始终返回相同的元素序列，那么 retry() 无法正常运行。

```
risky().cached().retry() //有问题的
```

如果 risky() 发布了一次错误，那么它将会持续发布错误，无论你重新订阅多少次均如此。为了解决这个问题，我们可以通过 defer() 进一步延迟 Observable 的创建。

```
Observable
    .defer(() -> risky())
    .retry()
```

即便从 risky() 返回的 Observable 得到缓存，defer() 也会多次调用 risky()。这样，我们每次可能会得到一个新的 Observable。

通过延迟和限制尝试进行重试

简单的 retry() 方法非常有用，但是没有节流或者不限制尝试次数的盲目重订阅是非常危险的。这样，CPU 或网络很快就会饱和，造成大量的负载。基本上，无参的 retry() 就像 while 循环包含了一个 try 语句，而后面的 catch 语句是空的。首先，样例应该限制尝试次数，该功能恰好就是内置的。

```
risky()
    .timeout(1, SECONDS)
    .retry(10)
```

retry() 的整型参数指定了要重新订阅的次数，因此 retry(0) 等价于根本不进行重试。如果上游的 Observable 失败了 10 次，那么最后的异常将会传递至下游。更灵活版本的 retry() 允许根据尝试次数和实际的异常，自行判断如何进行重试。

```
risky()
    .timeout(1, SECONDS)
    .retry((attempt, e) ->
        attempt <= 10 && !(e instanceof TimeoutException))
```

这个版本不仅将重新订阅的次数限制为 10，还会在出现 TimeoutException 异常时，永久放弃重试。

如果失败是暂时的，在尝试重新订阅之前等待一会儿是不错的办法。retry() 操作符并没有提供这种开箱即用的功能，但是它相对容易实现。retry() 有一个更健壮的版本，即 retryWhen()。它接收一个函数，这个函数能够处理由失败组成的 Observable。每次上游失败，这个 Observable 就会发布一个 Throwable。如下的代码片段想要在重试（这是它得名的原因）时转换 Observable，使其能够发布一个任意的事件。

```
risky()
    .timeout(1, SECONDS)
    .retryWhen(failures -> failures.delay(1, SECONDS))
```

这个 retryWhen() 样例接收了一个 Observable，后者会在上游失败时，发布一个 Throwable。样例只是简单地延迟了 1 秒，所以它会在 1 秒之后出现在最终的流中。此时，应该使用 retryWhen() 进行重试。如果只是返回相同的流（retryWhen(x -> x)），retryWhen() 的行为将和 retry() 完全一样，也就是在出现错误时立即重新订阅。借助 retryWhen()，我们还可以很容易地模拟 retry(10)（几乎是相同的，请继续阅读）。

```
.retryWhen(failures -> failures.take(10))
```

每次失败发生时，我们就会接收到一个事件。这里返回的流会在我们想要进行重试的时候，发布任意一个事件。因此，样例只转发前 10 个失败事件，这样每次失败后都会立即重试。但是，failures Observable 出现第 11 个失败时会怎样呢？这就是比较麻烦的地方了。在第 10 次失败之后，take(10) 操作符将会立即发布一个 onComplete 事件。因此，在第 10 次重试之后，retryWhen() 会接收到一个完成事件。这个事件会被解读为停止重试的信号，让下游进入完成状态。这意味着，在失败重试 10 次之后，样例将不再发布任何内容并进入完成状态。但是，如果 retryWhen() 返回的 Observable 出现错误，那么这个错误将会被传递至下游。

换句话说，只要 retryWhen() 中的 Observable 发布事件，它们就会被解读为重试请求。但是，如果发布完成或错误通知，那么样例将放弃重试，完成或错误通知会被传递至下游。failures.take(10) 会重试 10 次，在此之后如果出现了其他失败，样例不会传递最后的错误，而是将这个流成功完成。如下所示。

```
static final int ATTEMPTS = 11;

//...

.retryWhen(failures -> failures
        .zipWith(Observable.range(1, ATTEMPTS), (err, attempt) ->
                attempt < ATTEMPTS ?
                        Observable.timer(1, SECONDS) :
                        Observable.error(err))
        .flatMap(x -> x)
)
```

这看上去相当复杂，但是它的功能非常强大。样例将失败和从 1 到 11 的序列通过 zip 压缩在一起。我们最多只想执行 10 次重试，所以如果重试序列的值小于 11，样例会返回

timer(1, SECONDS)。retryWhen() 操作符会捕获这个事件，并在失败发生的 1 秒之后进行重试。但是，当第 10 次重试出现失败的时候，样例会返回包含该错误的 Observable，以最后一个看到的异常完成该重试机制。

这提供了很大的灵活性。我们可以在出现特定异常或者认为重试次数过多时，停止进行重试。另外，还可以调整多次尝试之间的延迟时间。例如，第一次重试可以立即进行，后续的重试间隔则呈指数级增加。

```
.retryWhen(failures -> failures
    .zipWith(Observable.range(1, ATTEMPTS),
        this::handleRetryAttempt)
    .flatMap(x -> x)
)

//...

Observable<Long> handleRetryAttempt(Throwable err, int attempt) {
    switch (attempt) {
        case 1:
            return Observable.just(42L);
        case ATTEMPTS:
            return Observable.error(err);
        default:
            long expDelay = (long) Math.pow(2, attempt - 2);
            return Observable.timer(expDelay, SECONDS);
    }
}
```

进行第一次重试的时候，样例返回的 Observable 会立即发布一个任意的事件，所以重试会立即执行。在这里，返回的事件的类型和值并不重要（只有发布时机真正有用），所以 42 可以替换成任意其他的值。最后一次重试时，样例向下游 Subscriber 转发一个异常，包含了最后失败的原因。对于第 2 次到第 10 次的重试，我们使用如下的指数公式进行计算。

$$delay(attempt) = \begin{cases} 0 & \text{if } attempt = 1 \\ 2^{attempt-2} & \text{if } attempt\{2,3,4,\cdots,10\} \end{cases}$$

7.2 测试和调试

流组合可能会非常困难，尤其是涉及时间的时候。比较令人开心的是，RxJava 对单元测试提供了良好的支持。使用 TestSubscriber 可以断言已发布的事件，但更重要的是，RxJava 有一个虚拟时间（virtual time）的概念。本质上，我们能够控制时间的流逝，所以依赖于时间的测试既快捷又可预测。

7.2.1 虚拟时间

在处理的所有应用程序中，时间都是一个重要的因素，这里讨论的并不是延迟和响应时间。所有事情的发生都有一个时间点，事件发生的顺序非常重要，而任务会调度到未来执行。因此，我们需要花费大量时间寻找只在特定日期或时区出现的 bug。在测试时间相

关的代码时，并没有什么既成的方法。有一种实践叫作**基于属性的测试**（property-based testing），它的目标是生成数百个测试用例（有时是随机的），以便于测试范围广泛的用户参数。例如，验证一个简单的属性：对于任何给定的日期，先增加后减少一个月将返回相同的日期，如下所示。

```
import spock.lang.Specification
import spock.lang.Unroll

import java.time.LocalDate
import java.time.Month

class PlusMinusMonthSpec extends Specification {

    static final LocalDate START_DATE =
            LocalDate.of(2016, Month.JANUARY, 1)

    @Unroll
    def '#date +/- 1 month gives back the same date'() {
        expect:
            date == date.plusMonths(1).minusMonths(1)
        where:
            date << (0..365).collect {
                day -> START_DATE.plusDays(day)
            }
    }

}
```

使用 Groovy 语言的 Spock 框架快速生成 366 个不同的测试用例。`expect` 块中的代码会针对 `where` 代码块生成的每个值都执行一遍。在 `where` 代码块中，我们迭代从 `0` 到 `365` 的整数值，从而生成 `2016-01-01` 到 `2016-12-31` 的所有日期。断言非常简单直接：如果将任意的日期增加一个月再减少一个月，那么我们能够再次得到之前的日期。不过，在 366 个测试用例中有 6 个是失败的。

```
date == date.plusMonths(1).minusMonths(1)
|    |  |     |              |
|    |  |     2016-02-29     2016-01-29
|    |  2016-01-30
|    false
2016-01-30


date == date.plusMonths(1).minusMonths(1)
|    |  |     |              |
|    |  |     2016-02-29     2016-01-29
|    |  2016-01-31
|    false
2016-01-31


date == date.plusMonths(1).minusMonths(1)
|    |  |     |              |
|    |  |     2016-04-30     2016-03-30
```

```
|    |  2016-03-31
|    false
2016-03-31

...
```

相信你能够计算出其他测试失败的日期。这个有点牵强的样例是为了展现时域（time domain）的复杂性。但是，日历的特殊性并不是我们在计算机系统中处理时间时感到头疼的根源。RxJava 通过尽可能地避免使用状态和采用纯函数，来解决并发的复杂性。这里的“纯”指函数（或操作符）应该显式声明所有的输入和输出。这使得测试更加容易。但是，对时间的依赖基本上都是隐形的。每次看到 `new Date()`、`Instant.now()`、`System.currentTimeMillis()` 等，我们就会对随时间变化的外部值产生依赖。依赖单例是不好的设计，从测试的角度更是如此。但是，读取当前时间实际上依赖于到处皆可访问的系统级单例。

为了突显对时间的依赖，有种模式是使用一个伪造的系统时钟。这对所有开发人员的要求都非常严格，要将时间相关的代码代理给一个可以仿造的特定服务。Java 8 通过引入 `Clock` 抽象规范了这种方法，大致如下所示。

```
public abstract class Clock {

    public static Clock system(ZoneId zone) { /* ... */ }

    public long millis() {
        return instant().toEpochMilli();
    }

    public abstract Instant instant();

}
```

有意思的是，RxJava 有一种类似的抽象，本书已经非常详细地介绍过了：`Scheduler`（参见 4.9.1 节）。你可能会问，`Scheduler` 与时间的流逝又有什么关系呢？因为 RxJava 中的所有事情要么立即发生，要么安排在未来某个时间发生。在 RxJava 中，正是 `Scheduler` 完全控制每行代码何时执行。

7.2.2　单元测试中的调度器

各种 `Scheduler`（比如 `io()` 或 `computation()`）的功能就是在特定的时间点运行任务。但是，有一个特殊的 `test()` `Scheduler`，它有两个非常有趣的方法：`advanceTimeBy()` 和 `advanceTimeTo()`。`TestScheduler` 的两个方法能够手动向前推进时间，否则，时间会被永远冻结。这意味着，在手动向前推动时间（可以在任何觉得有用的时间点）之前，这个 `Scheduler` 不会调度执行任何未来的任务。

以下样例是随着时间推移出现的事件序列。

```
TestScheduler sched = Schedulers.test();
Observable<String> fast = Observable
    .interval(10, MILLISECONDS, sched)
```

```
        .map(x -> "F" + x)
        .take(3);
    Observable<String> slow = Observable
        .interval(50, MILLISECONDS, sched)
        .map(x -> "S" + x);

    Observable<String> stream = Observable.concat(fast, slow);
    stream.subscribe(System.out::println);
    System.out.println("Subscribed");
```

在订阅之时，应该很快就会看到三个事件 F0、F1 和 F2，每个事件的延迟是 10 毫秒，随后是无限数量的事件 S0、S1...，每个事件间隔 50 毫秒。我们该如何测试这些流已经成功合并到一起，这些事件会按照正确的顺序在正确的时间出现？这里的关键在于显式传递使用 TestScheduler。

```
TimeUnit.SECONDS.sleep(1);
System.out.println("After one second");
sched.advanceTimeBy(25, MILLISECONDS);

TimeUnit.SECONDS.sleep(1);
System.out.println("After one more second");
sched.advanceTimeBy(75, MILLISECONDS);

TimeUnit.SECONDS.sleep(1);
System.out.println("...and one more");
sched.advanceTimeTo(200, MILLISECONDS);
```

程序的输出是可预测和可重复的，完全独立于系统时间、瞬间流量峰值、GC 暂停等。

```
Subscribed
After one second
F0
F1
After one more second
F2
S0
...and one more
S1
S2
```

下文描述了程序执行的过程。

(1) 订阅 stream Observable 之后，它会调度未来 10 毫秒后的 F0 任务。但是，这里使用了 TestScheduler，它会处于绝对的空闲状态，除非手动向前推进时间。
(2) 休眠 1 秒其实无关紧要，甚至可以省略，TestScheduler 独立于系统时间，因此根本不会有事件发布出来。这里的休眠只是为了证明 TestScheduler 能够正常运行。如果这里不是 TestScheduler，而是普通的（默认的）调度器，我们可以预期在控制台中已经出现了多个事件。
(3) 调用 advanceTimeBy(25ms) 会强制时间向前推进至第 25 毫秒，调度的任务会被触发或执行。这会导致事件 F0（第 10 毫秒）和 F1（第 20 毫秒）出现在控制台中。

(4) 接下来休眠的 1 秒内不会有任何输出，`TestScheduler` 会忽略真正的时间。但是，调用 `advanceTimeBy(75ms)`（所以现在的逻辑时间是第 100 毫秒）会进一步触发 `F2`（第 30 毫秒）和 `S0`（第 80 毫秒）事件。除此之外，不会发生其他事件。
(5) 在真实时间又消耗 1 秒之后，将时间的绝对值推进至 200 毫秒（`advanceTimeTo(200ms)`，而 `advanceTimeBy()` 使用的是相对时间）。`TestScheduler` 实现了此时触发 `S1`（第 130 毫秒）和 `S2`（第 180 毫秒）。但是除此之外不会触发其他事件，即便永远等待下去也是如此。

可以看到，`TestScheduler` 比普通的仿造 `Clock` 抽象更加智能。不仅能够完全控制当前时间，还能任意推迟所有事件。需要注意的是，基本上要将 `TestScheduler` 传递给每个接收可选 `Scheduler` 参数的操作符。为了简便，这些操作符会使用一个默认的 `computation()` `Scheduler`。但是从可测试性的角度，样例应该优先传递显式的 `Scheduler`。另外，可以考虑依赖注入的方式，在外部提供 `Scheduler`。

但是，只有 `TestScheduler` 还不够。它在单元测试中非常有用，单元测试的可预测性是必备的，偶尔出现的测试失败会令人非常沮丧。第 8 章将介绍支持异步 `Observable` 单元测试的工具和技术。

7.3 单元测试

长期以来，编写可测试的代码并拥有一套稳定的测试集都是必备的，并不是什么新颖的方式。无论以测试驱动开发（Test-Driven Development，TDD）精神优先编写测试，还是随后通过集成测试验证假设，我们都应该非常熟悉自动化测试了。因此，使用的工具（框架、库、平台）必须支持自动化测试，这种能力是在进行技术决策时应该考虑的一个方面。不用担心，尽管异步、事件驱动架构领域非常复杂，但是 RxJava 为单元测试提供了良好的支持。时间方面的确定性，再加上对纯函数和函数组合（良好的函数式编程基础）的关注能够极大地提升测试体验。

校验已发布的事件

首先，需要定义测试 `Observable` 的目标。如果方法返回 `Observable`，我们可能需要确保如下事项。

- 事件按照正确的顺序发布。
- 错误得到恰当地传递。
- 各种操作符的组合符合预期。
- 事件在恰当的时间出现。
- 支持回压。

除此之外，还有很多要求。上述前两项要求很简单，并不需要 RxJava 的特殊支持。基本上将发布的所有内容收集起来，然后使用我们喜欢的库执行断言即可。

```
import org.junit.Test;
import static org.assertj.core.api.Assertions.assertThat;
```

```
@Test
public void shouldApplyConcatMapInOrder() throws Exception {
    List<String> list = Observable
        .range(1, 3)
        .concatMap(x -> Observable.just(x, -x))
        .map(Object::toString)
        .toList()
        .toBlocking()
        .single();

    assertThat(list).containsExactly("1", "-1", "2", "-2", "3", "-3");
}
```

上述简单测试案例通过熟知的 toList() → toBlocking() → single() 构造，将 Observable <Integer> 转换成了 List<Integer>（参见 4.2 节）。正常情况下，Observable 都是异步的，为了具备可预测性并实现快捷的测试，样例必须要执行这样的转换。在使用 BlockingObservable 的时候，样例也可以很容易地断言 onError() 通知。异常会在订阅时重新简单地抛出。注意，检查型异常会使用 RuntimeException 进行包装——只有好的测试才能证明这一切。

```
import com.google.common.io.Files;
import static java.nio.charset.StandardCharsets.UTF_8;
import static org.assertj.core.api.Assertions.failBecauseExceptionWasNotThrown;


File file = new File("404.txt");
BlockingObservable<String> fileContents = Observable
    .fromCallable(() -> Files.toString(file, UTF_8))
    .toBlocking();

try {
    fileContents.single();
    failBecauseExceptionWasNotThrown(FileNotFoundException.class);
} catch (RuntimeException expected) {
    assertThat(expected)
        .hasCauseInstanceOf(FileNotFoundException.class);
}
```

如果想要以延迟执行的方式创建最多只发布一个元素的 Observable，那么 fromCallable() 操作符是非常便利的。它还会进行错误处理和回压，所以对于单元素的流，应该优先使用它，而不是 Observable.create()。使用其他类型的单元测试可以判断我们对各种操作符及其行为的理解是否正确。例如，concatMapDelayError() 到底是做什么的？你可以逐一进行尝试，但是拥有一个人人都能阅读和快速理解的自动化测试，会带来巨大的优势。

```
import static rx.Observable.fromCallable;

Observable<Notification<Integer>> notifications = Observable
    .just(3, 0, 2, 0, 1, 0)
    .concatMapDelayError(x -> fromCallable(() -> 100 / x))
    .materialize();
```

```
List<Notification.Kind> kinds = notifications
    .map(Notification::getKind)
    .toList()
    .toBlocking()
    .single();

assertThat(kinds).containsExactly(OnNext, OnNext, OnNext, OnError);
```

如果使用标准的 concatMap()，第二个元素（0）的转换会失败并终结整个流。但是，我们清晰地看到最终的流有 4 个元素：三个 OnNext，随后是 OnError。可以使用另外一个断言判断最终的值将是 33（100/3）、50 和 100。这很好地阐述了 concatMapDelayError() 是如何运行的：如果转换过程中出现了错误，错误不会向下游传递，而操作符会继续执行。只有上游源完成的时候，才会传递一个 onError 通知，表明在这个过程中遇到的错误。最后的这个测试用例已经无法将 Observable 转换成 List 了，因为它会立即抛出一个错误。在这种情况下，materialize() 就非常有用了：每种类型的事件（onNext、onCompleted 和 onError）都会使用一个通用的 Notification 对象进行包装，随后检查这些对象。但是这种方式非常烦琐，代码也难以阅读。这时使用 TestSubscriber 就非常便利了，如下所示。

```
Observable<Integer> obs = Observable
        .just(3, 0, 2, 0, 1, 0)
        .concatMapDelayError(x -> Observable.fromCallable(() -> 100 / x));

TestSubscriber<Integer> ts = new TestSubscriber<>();
obs.subscribe(ts);

ts.assertValues(33, 50, 100);
ts.assertError(ArithmeticException.class); //失败
```

TestSubscriber 类非常简单：它会在内部存储所有接收到的事件。这样，我们随后可以进行查询。TestSubscriber 还提供了一组断言，在测试场景下它们非常有用。我们需要做的就是创建一个 TestSubscriber，订阅要测试的 Observable 并检查它的内容。奇怪的是，上述测试会失败。assertError() 失败的原因在于预期流完成时将会出现 ArithmeticException，但实际得到的是 CompositeException。后者将这个过程中遇到的三个 ArithmeticException 都聚合了起来。这也从另一个方面说明了，通过实际运行操作符和执行自动化测试来学习操作符是非常有用的。

将 TestSubscriber 和 TestScheduler 组合使用时尤为高效。一个典型的场景是在向前推进时间的同时插入断言，以观察事件如何随时间的推移而流动。假设有一个返回 Observable 的服务，如下所示，它的实现细节无关紧要。

```
interface MyService {
    Observable<LocalDate> externalCall();
}
```

这里不会将各种关注项混合在一起，而是决定围绕 MyService 构建一个包装器。无论底层的 MyService 实现是什么，都能为其添加超时功能。基于你可能已经猜到的原因，以下代码更进一步，将 timeout() 操作符使用的 Scheduler 进行了外部化。

```
class MyServiceWithTimeout implements MyService {

    private final MyService delegate;
    private final Scheduler scheduler;

    MyServiceWithTimeout(MyService d, Scheduler s) {
        this.delegate = d;
        this.scheduler = s;
    }

    @Override
    public Observable<LocalDate> externalCall() {
        return delegate
                .externalCall()
                .timeout(1, TimeUnit.SECONDS,
                    Observable.empty(),
                    scheduler);
    }
}
```

MyServiceWithTimeout 包装了另外一个 MyService 实例，添加了具有备用功能的 1 秒超时。按照 RxJava 的理念，每个类都有一个可组合的功能，就像操作符非常集中又易于组合。假设我们想要测试超时功能能否真正发挥作用，单元测试应该极其快捷。7.2.1 节的开篇介绍了 PlusMinusMonthSpec，为 21 世纪的每一天（超过 36 000 个测试用例）调用一遍该测试大约需要 1 秒。一个好的单元测试，执行时间不应该超过几毫秒。

1 秒的超时似乎并不是太长，但是如果有上百个这样的场景，那么就会消耗相当长时间了。我们可以将这个超时时间外部化（无论如何都是一种好的做法），并在单元测试中缩短它的值，比如变成 100 毫秒。在这样的情况下，样例可以休眠 90 毫秒，并断言超时机制没有发挥作用；然后再休眠 20 秒，校验超时机制是否返回了一个空的 Observable。令人遗憾的是，这种方式非常脆弱，可能会受到上下文切换、垃圾收集暂停、系统负载变化等因素的影响。简而言之，测试可以相对稳定，也可以相对快捷。但是，它越快，出现误报失败的频率就越高。不稳定的测试比根本没有测试更糟糕，因为这令人感到沮丧，你无法信任它们，并最终将其移除。

RxJava 提供了一种人为的、可控的时钟，它是完全可预测的。通过手动向前推进时间，我们能够实现百分之百的精确性，同时又能实现测试的快捷执行。首先，仿造一个 MyService（通过 Mockito），它能返回任意的 Observable。

```
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.mock;

private MyServiceWithTimeout mockReturning(
            Observable<LocalDate> result,
            TestScheduler testScheduler) {
    MyService mock = mock(MyService.class);
    given(mock.externalCall()).willReturn(result);
    return new MyServiceWithTimeout(mock, testScheduler);
}
```

现在，编写两个单元测试。第一个测试能够确保，如果 externalCall() 永远不结束，样例

在 1 秒之后会遇到超时。

```
@Test
public void timeoutWhenServiceNeverCompletes() throws Exception {
    //given
    TestScheduler testScheduler = Schedulers.test();
    MyService mock = mockReturning(
            Observable.never(), testScheduler);
    TestSubscriber<LocalDate> ts = new TestSubscriber<>();

    //when
    mock.externalCall().subscribe(ts);

    //then
    testScheduler.advanceTimeBy(950, MILLISECONDS);
    ts.assertNoTerminalEvent();
    testScheduler.advanceTimeBy(100, MILLISECONDS);
    ts.assertCompleted();
    ts.assertNoValues();
}
```

`never()` 操作符返回的 `Observable` 永远不会完成，并且不会发布任何值。这模拟了 `MyService` 调用非常缓慢的场景。然后，进行两个断言的序列。首先，将时间向前推移至超时阈值（950 毫秒）之前，并确保 `TestSubscriber` 没有完成或失败。在 100 毫秒（超时阈值）之后，样例断言流已经完成（`assertCompleted()`），并且没有任何值（`assertNoValues()`）。我们也可以使用 `assertError()`。

第二个测试要确保在达到配置的阈值之前，超时机制不会发挥作用。

```
@Test
public void valueIsReturnedJustBeforeTimeout() throws Exception {
    //given
    TestScheduler testScheduler = Schedulers.test();
    Observable<LocalDate> slow = Observable
            .timer(950, MILLISECONDS, testScheduler)
            .map(x -> LocalDate.now());
    MyService myService = mockReturning(slow, testScheduler);
    TestSubscriber<LocalDate> ts = new TestSubscriber<>();

    //when
    myService.externalCall().subscribe(ts);

    //then
    testScheduler.advanceTimeBy(930, MILLISECONDS);
    ts.assertNotCompleted();
    ts.assertNoValues();
    testScheduler.advanceTimeBy(50, MILLISECONDS);
    ts.assertCompleted();
    ts.assertValueCount(1);
}
```

`advanceTimeBy()` 在测试中相当于休眠，它会等待一些操作执行，但是并没有真正地休眠。你可以测试各种类型的操作符，比如 `buffer()`、`sample()` 等，只要精心传递自定义

的 Scheduler 即可。谈到调度器，我们很容易使用 Schedulers.immediate()（参见 4.9.1 节），而不是标准的调度器。但是，这个 Scheduler 不支持同步，它会在调用者线程的上下文中执行各种操作。这种方式在某些场景下能够正常运行，但是一般应该优先使用 TestScheduler，它的用例更广泛。

遵循依赖注入原则是非常重要的。否则，我们就无法将各种 Scheduler 与 TestScheduler 进行替换了。在这方面，有些技术可以提供帮助，比如 RxJavaSchedulersHook 插件（plug-in）。RxJava 有一组插件，它们能够全局地改变库的行为。举例来说，RxJavaSchedulersHook 能够将标准的 computation() Scheduler（或其他调度器）替换为测试调度器，如下所示。

```
private final TestScheduler testScheduler = new TestScheduler();

@Before
public void alwaysUseTestScheduler() {
    RxJavaPlugins
        .getInstance()
        .registerSchedulersHook(new RxJavaSchedulersHook() {
            @Override
            public Scheduler getComputationScheduler() {
                return testScheduler;
            }

            @Override
            public Scheduler getIOScheduler() {
                return testScheduler;
            }

            @Override
            public Scheduler getNewThreadScheduler() {
                return testScheduler;
            }
        });
}
```

这种全局的方式有很多局限性。在整个 JVM 中，我们只能注册 RxJavaSchedulersHook 一次，所以第二次调用这个 @Before 方法将会失败。你可以进行处理，但是这会变得更加复杂。同时，并行运行单元测试（通常情况下，单元测试之间是独立的，所以这并不是什么问题）也不可能实现。因此，控制时间的唯一可扩展方案就是尽可能地显式传递 TestScheduler。

TestSubscriber 的最后一个练习就是测试回压。6.2.4 节研究了生成连续自然数的两个无穷 Observable 实现，其中一个使用老式未经加工的 Observable.create()，它不支持回压。

```
Observable<Long> naturals1() {
    return Observable.create(subscriber -> {
        long i = 0;
        while (!subscriber.isUnsubscribed()) {
            subscriber.onNext(i++);
        }
    });
}
```

以下是更高级且推荐的实现方式，完全支持回压。

```
Observable<Long> naturals2() {
    return Observable.create(
        SyncOnSubscribe.createStateful(
            () -> 0L,
            (cur, observer) -> {
                observer.onNext(cur);
                return cur + 1;
            }
    ));
}
```

从功能的角度，这两者相同，都是无穷流，但是（比如）你可以只选取其中一个子集。不过，借助 TestSubscriber，可以很容易地以单元测试的方式校验给定的 Observable 是否支持回压。

```
TestSubscriber<Long> ts = new TestSubscriber<>(0);

naturals1()
        .take(10)
        .subscribe(ts);

ts.assertNoValues();
ts.requestMore(100);
ts.assertValueCount(10);
ts.assertCompleted();
```

这个样例的关键部分是 TestSubscriber<>(0)。如果没有它，TestSubscriber 只会以源规定的速度接收所有内容。但是，如果在订阅之前没有请求任何数据，那么 TestSubscriber 也不会从 Observable 中请求数据。这就是尽管源 Observable 已经发布了 10 个值，assertNoValues() 依然能够成功的原因。随后请求 100 个条目（为了安全起见），但是源 Observable 只生成了 10 个条目，这就是它能够生成的数量。对于 naturals1 的测试几乎会立即失败，输出的消息如下。

```
AssertionError: No onNext events expected yet some received: 10
```

原生的 Observable 会在接收 10 个事件后就停止发布事件，尽管这个 Observable 可能是无穷的。take(10) 操作符会立即取消订阅，结束内部的 while 循环。但是，naturals1 忽略了 TestSubscriber 的回压请求，后者会接收到它没有请求的条目。如果将源替换为 naturals2，测试就能够通过。这也是我们避免使用 Observable.create()，而使用内置的工厂和 SyncOnSubscribe 的另一个原因。

TestSubscriber 有很多其他的断言。其中有一些会阻塞等待完成，比如 awaitTerminalEvent()。但是大多数会断言订阅者当前的状态，所以你可以随时间推移观察事件的流动。

7.4 监控和调试

监控各种流之间的交互并在出现问题时进行排查解决，在 RxJava 中是一个非常困难的主题。实际上，相对于阻塞式架构，每种异步事件驱动架构在进行问题排查时都更加困难。

同步操作失败时，异常会在整个调用堆栈流动，暴露导致问题的精确操作序列，从 HTTP 服务器到所有的过滤器、切面、业务逻辑等。在异步系统中，调用堆栈的用处很有限，因为事件跨越线程边界后，就无法获取原始的调用堆栈了。相同的情况也适用于分布式系统。本节会就如何在使用 RxJava 的应用程序中更容易地监控和调试，给出一定的提示。

7.4.1 doOn...()回调

每个 Observable 都有一组回调方法，你能够使用它们查看各种事件，几种方法如下。

- doOnCompleted()
- doOnEach()
- doOnError()
- doOnNext()
- doOnRequest()
- doOnSubscribe()
- doOnTerminate()
- doOnUnsubscribe()

它们的共同点就是不允许以任何方式改变 Observable 的状态。而且它们都返回同一个 Observable，于是，这些方法成为了嵌入日志逻辑的理想场所。例如，有些初学者很容易忘记，Observable.create() 中的代码会为每个新的 Subscriber 都执行一遍。这一点非常重要，当订阅引发像网络调用之类的副作用时更是如此。为了检测这样的问题，最好记录下对重要源的每次订阅。

```
Observable<Instant> timestamps = Observable
    .fromCallable(() -> dbQuery())
    .doOnSubscribe(() -> log.info("subscribe()"));

timestamps
    .zipWith(timestamps.skip(1), Duration::between)
    .map(Object::toString)
    .subscribe(log::info);
```

上述程序会查询数据库（dbQuery()）并以 Observable<Instant> 的形式检索一些时序数据。我们想要对这个流进行一些转换，计算出每组连续 Instant 之间的持续时间（使用 java.time 包的 Duration 类）：第一个和第二个之间、第二个和第三个之间，以此类推。实现该功能的一种方式就是使用 zip() 将该流与它本身进行组合，并偏移一个元素。通过这种方式，将第一个元素和第二个元素连接起来，第二个元素和第三个元素连接起来，直至最后。没有预料到的是，zipWith() 实际上会订阅所有底层流，这意味着同一个 timestamps Observable 订阅了两次。这个问题通过观察 doOnSubscribe() 就可以发现，因为这个方法被调用了两次。这会导致重复的数据库查询，这个问题在第 2 章已经进行过详细地讨论。

谈到 zip()，借助回压功能，它不再无限缓存较快的流，等待较慢的流发布事件。相反，它会从每个 Observable 中请求固定数量的一批值。如果它接收到更多的值，将抛出 MissingBackpressureException。

```
    .doOnSubscribe(() -> log.info("subscribe()"))
    .doOnRequest(c -> log.info("Requested {}", c))
    .doOnNext(instant -> log.info("Got: {}", instant));
```

doOnRequest() 会记录 Requested 128，这个值是由 zip 操作符选择的。即便源是无穷的，或者事件的数量非常庞大，如果 Observable 对回压支持良好，随后最多也只能看到 128 条信息，比如 Got: ...。doOnNext() 是我们可以使用的另一个回调。还有 doOnError() 也经常会用到，每当上游出现错误通知，就会触发这个回调。你不能使用 doOnError() 进行任何错误处理，它只能用来进行日志记录。这个回调不会消费错误通知，而是将其向下游继续传递。

```
Observable<String> obs = Observable
    .<String>error(new RuntimeException("Swallowed"))
    .doOnError(th -> log.warn("onError", th))
    .onErrorReturn(th -> "Fallback");
```

onErrorReturn() 看上去非常整洁。它能够很轻松地吞噬异常，将异常替换为一个备用值。但是记录异常是我们的责任。为了让函数更加短小且可组合，样例首先在 doOnError() 中记录异常，然后在下一行代码中进行无声地处理，这种方式会更健壮一些。不对异常进行日志记录可不是什么好主意，我们必须要仔细决策，不可疏忽。

其他的操作符基本上都不言自明，不过以下这组有点例外。

- doOnEach()

  这个操作符可以为每个 Notification 所调用，即 onNext()、onCompleted() 和 onError()。它既可以接收每个 Notification 调用的 lambda 表达式，也可以接收一个 Observer。

- doOnTerminate()

  出现 onCompleted() 或 onError() 时，都会触发该回调。两者无法进行区分，所以最好单独使用 doOnCompleted() 或 doOnError()。

7.4.2 测量和监控

回调不仅用于日志记录。如果应用程序内置各种遥测探针（比如简单的计数器、定时器、分布直方图等），并且能够在外部进行访问，那么这会极大地减少问题排查的时间，我们也能很好地观察应用程序正在做什么。在简化指标的收集和发布方面，有很多可用的库，比如 Dropwizard metrics。在使用这个库之前，需要一些初始搭建工作，如下所示。

```
import com.codahale.metrics.MetricRegistry;
import com.codahale.metrics.Slf4jReporter;
import org.slf4j.LoggerFactory;


MetricRegistry metricRegistry = new MetricRegistry();
Slf4jReporter reporter = Slf4jReporter
    .forRegistry(metricRegistry)
    .outputTo(LoggerFactory.getLogger(SomeClass.class))
    .build();
reporter.start(1, TimeUnit.SECONDS);
```

MetricRegistry 是各种指标的一个工厂。另外，还搭建了一个 Slf4jReporter，它会将当前的统计快照推送至给定的 SLF4J logger。除此之外，还有推送至 Graphite 和 Ganglia 的 reporter。简单配置完成之后，我们就可以监控自己的系统了。

我们能想到的最简单的指标就是 Counter，它可以递增或递减。你可以用它来测量通过流的事件数量，如下所示。

```
final Counter items = metricRegistry.counter("items");
observable
        .doOnNext(x -> items.inc())
        .subscribe(...);
```

订阅这个 Observable 之后，Counter 就会显示目前已经生成了多少条目。如果样例将这个信息推送至外部的监控服务器，如 Graphite，并随着时间推移在图表上展现，这个信息会更加有用。

我们想要捕获的另一个重要指标，就是有多少条目正在并发处理。例如，flatMap() 可以很容易地生成数百个甚至更多的并发 Observable，并同时订阅它们。如果我们能够知道有多少个这样的 Observable（考虑一下处于打开状态的数据库连接、WebSocket 等），那么就可以在很大程度上了解系统是如何运行的。

```
Observable<Long> makeNetworkCall(long x) {
    //...
}


Counter counter = metricRegistry.counter("counter");
observable
        .doOnNext(x -> counter.inc())
        .flatMap(this::makeNetworkCall)
        .doOnNext(x -> counter.dec())
        .subscribe(...);
```

当事件在上游出现时，样例递增这个计数器；事件在 flatMap() 后面出现（这意味着某个异步操作刚刚发布了内容）时，递减这个计数器。在空闲的系统中，计数器的值始终为零，但是，如果上游 observable 生成了大量的事件，而 makeNetworkCall() 相对比较慢，计数器的值就会飙升，清晰标明瓶颈的位置。

上述样例假设 makeNetworkCall() 始终只返回一个条目，并且永远不会失败（不会以 onError() 的形式完成该流）。如果你想要测量从开始订阅内部 Observable（工作开始执行的时间）到它完成的时长，那么也很简单。

```
observable
    .flatMap(x ->
        makeNetworkCall(x)
            .doOnSubscribe(counter::inc)
            .doOnTerminate(counter::dec)
    )
    .subscribe(...);
```

最复杂的指标之一是 Timer，它测量了两个时间点之间的时长。我无意夸大这个指标的价

值，但是借助它，我们可以测量网络调用延迟、数据库查询时间、用户响应时间等。测量时间的方式一般是获取当前时间的快照，执行一些耗时的操作，然后计算现在时间和之前时间的差值。这种方式在 Metrics 库中进行了如下封装。

```
import com.codahale.metrics.Timer;

Timer timer = metricRegistry.timer("timer");
Timer.Context ctx = timer.time();
//一些冗长的操作……
ctx.stop();
```

这个 API 会将操作的开始时间封装到 `Timer.Context` 中，并假设要测试的代码是阻塞式的。但是，如果我们想要测量从订阅一个无法控制的 `Observable` 到它结束的时长，又该怎么处理呢？在这里，依赖 `doOnSubscribe()` 和 `doOnTerminate()` 是不够的，因为无法在它们之间传递 `Timer.Context`。幸而，RxJava 通过一个额外的组合层解决了这个问题。

```
Observable<Long> external = //...

Timer timer = metricRegistry.timer("timer");

Observable<Long> externalWithTimer = Observable
        .defer(() -> Observable.just(timer.time()))
        .flatMap(timerCtx ->
                external.doOnCompleted(timerCtx::stop));
```

我们使用了一点小技巧。首先，使用 `defer()` 操作符延迟了启动时间。通过这种方式，定时器会在订阅发生时恰好启动。随后，以某种方式将 `Timer.Context` 实例替换为实际想要测试的 `Observable`（`external`）。但是，在返回 `external Observable` 之前，停止运行计时器。通过这种方式，你可以测量任何无法控制的 `Observable` 订阅和终止之间的时长。

如果你需要综合的、企业级的监控层，那么可以考虑使用 RxJava 编写的 Hystrix。这个库是第 8 章的一个研究案例（参见 8.2 节）。

7.5　小结

任何反应式库或框架的调试和问题排查都具有挑战性，这是它们的异步性和事件驱动架构决定的。RxJava 也不例外，但是它提供了一些工具，使得开发人员和运维人员的工作更加轻松。

- 首先，RxJava 拥抱错误，使其更容易处理和管理。
- 其次，它提供了一些设施，实时监控和调试流。
- 最后，它对单元测试提供了良好的支持。

实际上，对于时间敏感的操作符，能够控制系统时钟是极其有用的。起初，RxJava 的问题排查可能会比较困难。不过，它提供了清晰的 API 和严格的契约，而不是表面上看起来更简单的阻塞式代码。后者可能存在隐藏的竞态条件和较差的吞吐量问题。

第 8 章

案例学习

托马什·努尔凯维茨（Tomasz Nurkiewicz）

本章展示了使用 RxJava 编写实际应用程序的用例。Reactive Extensions 的 API 非常强大，但是首先必须要有 `Observable` 源。由于回压和 Rx 契约的限制，从头编写 `Observable` 可能是一件非常具有挑战性的事情。好消息是，很多库和框架已经原生支持 RxJava。同时，在一些具有异步特征的平台上，RxJava 是非常有用的。

通过本章的学习，你将了解 RxJava 是如何改善已有架构的设计并增强其功能的。我们还会探讨一些更高级的话题，这涉及反应式应用程序部署到生产环境，比如内存泄漏。当本章结束时，你应该相信 RxJava 是一个成熟和通用的框架，足以在真正的现代应用程序中实现各种用例。

8.1　使用RxJava进行Android开发

RxJava 在 Android 开发人员中非常流行。首先，图形化用户界面本质上是事件驱动的，因为事件来源于各种操作，比如键盘输入或鼠标移动。其次，与 Swing 和其他 GUI 环境类似，Android 对线程的要求非常苛刻。不应该阻塞 Android 主线程，以免造成用户界面的冻结，但是用户界面的所有更新必须在主线程中进行。这些问题会在 8.1.3 节中得到解决。但是，如果说在 Android 中使用 RxJava 只需要学习一件事，那么一定不要错过接下来的部分。本节将阐述内存泄漏的问题，以及如何轻松避免它们。

8.1.1　避免Activity中的内存泄漏

Android 特有的一个缺陷就是 `Activity` 相关的内存泄漏。这种问题的发生场景是 `Observer`

持有对任意 GUI 组件的强引用，而这个 GUI 组件又反过来引用了整个父 Activity 实例。当你切换屏幕界面或者按下返回按钮时，Android 就会销毁当前的 Activity 并最终对其进行垃圾回收。Activity 是非常大的对象，所以即时对它们进行清理是非常重要的。但是，如果 Observer 持有了对这种 Activity 的引用，它可能永远不会被垃圾回收，从而引起内存泄漏，最终设备会彻底终止这个应用程序。以下面的代码为例。

```
public class MainActivity extends AppCompatActivity {

    private final byte[] blob = new byte[32 * 1024 * 1024];

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView text = (TextView) findViewById(R.id.textView);
        Observable
                .interval(100, TimeUnit.MILLISECONDS)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(x -> {
                    text.setText(Long.toString(x));
                });
    }

}
```

这里的 blob 字段只是为了加快出现内存泄漏的效果，我们可以将 MainActivity 设想为一个相当复杂的对象树。这个简单的应用程序看上去并没有什么问题。每隔 100 毫秒，它就会使用当前计数器的值更新文本域。但是，如果你在设备上多次旋转屏幕，应用程序就会因为 OutOfMemoryError 而出现崩溃。这里发生的事情如下所示。

(1) 创建 MainActivity，在执行 onCreate() 的时候，订阅 interval()。
(2) 每隔 100 毫秒，使用当前计数器的值更新 text。可以先忽略 mainThread() 这个 Scheduler，8.1.3 节会对其进行介绍。
(3) 设备改变了方向。
(4) MainActivity 被销毁，继而创建一个新的 MainActivity，onCreate() 被再次执行。
(5) 目前有两个 Observable.interval() 在运行，因为没有取消对第一个 Observable 的订阅。

实际上，现在有两个 Observable.interval() 在运行，其中一个是已销毁 Activity 的残留物，但这还不是最糟糕的。interval() 操作符会使用一个后台线程（通过 computation() Scheduler）来发布计数器事件。这些事件随后会传递给 Observer，其中有个 Observer 会持有对 TextView 的引用，而 TextView 会反过来持有对旧 MainActivity 的引用。现在，发布 interval() 事件的线程成了新的 GC 根，所以它直接或间接引用的所有对象都不能进行垃圾回收。换句话说，即便第一个 MainActivity 实例销毁了，它也不能被垃圾回收，blob 占据的内存也无法进行释放。每次方向变化（或者每当 Android 决定要销毁特定的 Activity）都会增加内存的泄漏。解决方案其实非常简单：取消订阅，让 interval() 知道何时不再需要它（参见 2.3 节）。就像 onCreate() 一样，Android 也有一个销毁回调，名为 onDestroy()，如下所示。

```
private Subscription subscription;

@Override
protected void onCreate(Bundle savedInstanceState) {
    //...
    subscription = Observable
        .interval(100, TimeUnit.MILLISECONDS)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(x -> {
            text.setText(Long.toString(x));
        });
}

@Override
protected void onDestroy() {
    super.onDestroy();
    subscription.unsubscribe();
}
```

创建 Observable 的时候，是作为 Activity 生命周期的一部分。如果 Activity 销毁，需要确保取消对 Observable 的订阅。调用 unsusbcribe() 会解除 Observer 与 Observable 的关联，这样，就能进行垃圾回收了。现在，整个 MainActivity 连同 Observer 都可以进行回收了。同时，interval() 本身也不会再发布事件，因为没有人对其进行监听。实现了双赢的效果。

如果要在某个 Activity 中创建多个 Observable，那么持有对所有 Subscription 的引用可能会变得非常烦琐。在这种情况下，CompositeSubscription 是一个非常便利的容器。每个 Subscription 都可以插入 CompositeSubscription 中，在销毁的时候，一步就能轻松地对它们全部取消订阅，如下所示。

```
private CompositeSubscription allSubscriptions = new CompositeSubscription();

@Override
protected void onCreate(Bundle savedInstanceState) {
        //...
    Subscription subscription = Observable
        .interval(100, TimeUnit.MILLISECONDS)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(x -> {
            text.setText(Long.toString(x));
        });
    allSubscriptions.add(subscription);
}

@Override
protected void onDestroy() {
    super.onDestroy();
    allSubscriptions.unsubscribe();
}
```

值得一提的是，在任何环境下，如果不再使用某个 Observable，对其取消订阅都是一种好的实践。在资源有限的移动设备中，这一点显得尤为重要。既然你已经了解了 Android 中

内存管理的缺陷，接下来就可以重新设计我们的移动应用程序了。首先探讨 Retrofit，这是一个内置了 RxJava 支持的 HTTP 客户端，在移动环境中非常流行。

8.1.2 Retrofit对RxJava的原生支持

Retrofit 是一个发送 HTTP 请求的库，在 Android 生态系统中的使用尤为普遍。它并不局限于 Android，也不是 HTTP 客户端方案的唯一选择。但是，因为它对 RxJava 提供了支持，所以是移动应用程序很好的可选方案。它不仅适用于充分考虑到 RxJava 的场景，也适用于仅仅想要恰当处理 HTTP 代码的场景。在网络相关的代码中使用 RxJava 的优势，在于它能够在线程之间方便地进行切换。在体验 Retrofit 之前，我们需要添加如下依赖，包括库本身、针对 RxJava 的适配器以及针对 Jackson JSON 解析功能的转换器。

```
compile 'com.squareup.retrofit2:retrofit:2.0.1'
compile 'com.squareup.retrofit2:adapter-rxjava:2.0.1'
compile 'com.squareup.retrofit2:converter-jackson:2.0.1'
```

Retrofit 倡导以类型安全的方式与 RESTful 服务进行交互，它要求首先声明一个 Java 接口，但是不需要实现。这个接口随后会被透明地转换成一个 HTTP 请求。出于练习的目的，样例会与 Meetup API 进行交互，这是一个用于组织活动的流行服务，其中的一个端点能够返回指定位置附近的城市列表。

```
import retrofit2.http.GET;
import retrofit2.http.Query;


public interface MeetupApi {

    @GET("/2/cities")
    Observable<Cities> listCities(
        @Query("lat") double lat,
        @Query("lon") double lon
    );

}
```

Retrofit 将对 `listCities()` 的方法调用转换为网络调用。在内部，发起对 `/2/cities?lat=...&lon=...` 资源的 HTTP GET 请求。请注意返回类型。首先，我们已经有了强类型的 `Cities`，而不是使用 `String` 或弱类型的嵌套 Map（map-of-maps）。但是更重要的是，`Cities` 源于一个 `Observable`，响应到达的时候，`Observable` 会发布这个对象。`Cities` 类映射了在 JSON 中从服务器端接收到的大部分字段，这里省略了 getter 和 setter。

```
public class Cities {
    private List<City> results;
}

public class City {
    private String city;
    private String country;
    private Double distance;
    private Integer id;
```

```
        private Double lat;
        private String localizedCountryName;
        private Double lon;
        private Integer memberCount;
        private Integer ranking;
        private String zip;
    }
```

这样的方式很好地平衡了抽象（使用高层级的理念，如方法调用和强类型的响应）和底层细节（网络调用的异步性）。尽管 HTTP 具有请求—响应的语义，但是对不可避免的延迟使用 Observable 进行抽象，这样它就无法隐藏在糟糕的阻塞式 RPC（远程过程调用）抽象后面了。令人遗憾的是，为了与这个特殊的 API 进行交互，还需要配置很多胶水代码。具体的情况可能略有差异，重要的是了解正确解析 JSON 响应的步骤。

```
import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.PropertyNamingStrategy;
import retrofit2.Retrofit;
import retrofit2.adapter.rxjava.RxJavaCallAdapterFactory;
import retrofit2.converter.jackson.JacksonConverterFactory;


ObjectMapper objectMapper = new ObjectMapper();
objectMapper.setPropertyNamingStrategy(
    PropertyNamingStrategy.CAMEL_CASE_TO_LOWER_CASE_WITH_UNDERSCORES);
objectMapper.configure(
    DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);

Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.meetup.com/")
        .addCallAdapterFactory(
            RxJavaCallAdapterFactory.create())
        .addConverterFactory(
            JacksonConverterFactory.create(objectMapper))
        .build();
```

首先，需要优化一下 Jackson 库中的 ObjectMapper，从而无缝地将下划线的字段名转换为 Java Bean 使用的驼峰式命名约定，比如，JSON 中的 localized_country_name 转换为 City 类中的 localizedCountryName。其次，我们想要忽略 bean 类中没有进行映射的字段。JSON API 会进行演化，添加一些客户端尚未支持的新字段。比较合理的一种默认行为就是忽略掉这些字段，只使用对我们有意义的字段。这样，服务器端可以给响应添加新的字段，而不会破坏现有的客户端。

有了 Retrofit 之后，我们就可以合成 MeetupApi 实现了。在整个客户端代码中，都可以使用这个实现类。

```
MeetupApi meetup = retrofit.create(MeetupApi.class);
```

最后，借助 MeetupApi，就可以发送 HTTP 请求并发挥 RxJava 的威力了。接下来，构建一个更综合的样例。首先，使用 Meetup API 获取给定位置附近所有城镇的列表。

```
double warsawLat = 52.229841;
double warsawLon = 21.011736;
Observable<Cities> cities = meetup.listCities(warsawLat, warsawLon);
Observable<City> cityObs = cities
        .concatMapIterable(Cities::getResults);
Observable<String> map = cityObs
        .filter(city -> city.distanceTo(warsawLat, warsawLon) < 50)
        .map(City::getCity);
```

先使用 concatMapIterable() 扩展只包含一个条目的 Observable<Cities>，将找到的每个城市放到 Observable<City> 中。然后，过滤距离原始位置小于 50 公里的城市。最后，将城市的名称抽取出来。下一个目标是查询华沙附近每个城市的人口数量，看一下半径 50 公里内有多少人居住。为了实现这一点，我们必须查询 GeoNames 提供的另外一个 API。这个方法能够根据给定的名称查询位置，返回其人口。我们将再次使用 Retrofit 来连接这个 API。

```
public interface GeoNames {

    @GET("/searchJSON")
    Observable<SearchResult> search(
            @Query("q") String query,
            @Query("maxRows") int maxRows,
            @Query("style") String style,
            @Query("username") String username);

}
```

样例必须还要有一个 JSON 对象映射到数据对象（这里省略了 getter 和 setter）。

```
class SearchResult {
    private List<Geoname> geonames = new ArrayList<>();
}

public class Geoname {
    private String lat;
    private String lng;
    private Integer geonameId;
    private Integer population;
    private String countryCode;
    private String name;
}
```

实例化 GeoNames 的方式与 MeetupApi 类似。

```
GeoNames geoNames = new Retrofit.Builder()
        .baseUrl("http://api.geonames.org")
        .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
        .addConverterFactory(JacksonConverterFactory.create(objectMapper))
        .build()
        .create(GeoNames.class);
```

突然之间，示例应用程序使用了两个不同的 API，但是将它们统一组合了起来。查询 GeoNames API 获取每个城市的名称并抽取人口数量。

```
Observable<Long> totalPopulation = meetup
        .listCities(warsawLat, warsawLon)
        .concatMapIterable(Cities::getResults)
        .filter(city -> city.distanceTo(warsawLat, warsawLon) < 50)
        .map(City::getCity)
        .flatMap(geoNames::populationOf)
        .reduce(0L, (x, y) -> x + y);
```

如果你稍微思考一下，会发现上面的代码以非常简洁的方式做了大量的工作。首先，它请求 MeetupApi 获取城市组成的一个列表。随后，获取了每个城市的人口数量。接下来，借助 reduce() 对人口数量的响应（可能会异步返回）进行求和操作。最后，整个计算管道以 Observable<Long> 结束，所有城市的人口累积计算完成时，会发布出一个 long 类型的值。这展示了 RxJava 真正的威力，展现了不同来源的流如何无缝组合在一起。例如，populationOf() 方法实际上是一个非常复杂的操作符链，它会对 GeoNames 发起 HTTP 请求并根据城市名称抽取人口数量信息。

```
public interface GeoNames {

    default Observable<Integer> populationOf(String query) {
        return search(query)
            .concatMapIterable(SearchResult::getGeonames)
            .map(Geoname::getPopulation)
            .filter(p -> p != null)
            .singleOrDefault(0)
            .doOnError(th ->
             log.warn("Falling back to 0 for {}", query, th))
            .onErrorReturn(th -> 0)
            .subscribeOn(Schedulers.io());
    }

    default Observable<SearchResult> search(String query) {
        return search(query, 1, "LONG", "some_user");
    }

    @GET("/searchJSON")
    Observable<SearchResult> search(
        @Query("q") String query,
        @Query("maxRows") int maxRows,
        @Query("style") String style,
        @Query("username") String username
    );

}
```

search() 方法在代码底部采用默认方法进行了包装，所以它非常易于使用。在接收到以 JSON 包装的 SearchResult 对象之后，样例将其拆解为独立的搜索结果，确保在响应中不会出现缺少人口信息的情况。如果出现了错误，将会返回 0。最后，保证每个人口信息的查询都会在一个 io() 调度器中执行，以实现更好的并发性。在这里，subscribeOn() 实际上至关重要。如果没有它，每个城市对人口信息的请求都是序列化的，会极大地增加整体的延迟。但是，flatMap() 会对每个城市调用 populationOf() 方法，并在需要的时候进行订阅。这样，每个城市的数据都会并发获取。实际上，还可以为获取人口数据的请求添

加一个 timeout() 操作符，从而实现更好的响应时间，而代价是不完整的数据。如果没有 RxJava，实现这样的场景会需要大量手动线程池集成。即便使用 CompletableFuture（参见 5.4 节），这个任务也并不简单。但是，RxJava 凭借非侵入性并发和强大的操作符，能够很轻松地编写简洁且易于理解的代码。

将 Retrofit 驱动的两个不同 API 组合起来，运行效果非常好。我们还可以将完全不相关的 Observable 组合在一起，比如一个来源于 Retrofit，另一个来源于 JDBC 调用，还有一个接收的是来自 JMS 的消息。这些用例都很容易实现，它们既没有泄漏抽象，也没有提供关于底层流实现性质的过多细节。

8.1.3　Android中的调度器

每个 Android 开发人员可能都会犯的第一个错误就是阻塞 UI 线程。在 Android 中有一个指定的主线程，它会与用户界面（UI）进行双向交互。不仅原生组件的回调要在主线程中调用处理器，组件的更新（更新标签、绘制）也必须在主线程中。这种限制会极大地简化 UI 的内部架构，但是也有以下严重的缺点。

- 如果在回调处理中尝试进行耗时的操作（一般是阻塞式的网络调用），这个 UI 事件会阻碍其他 UI 事件的处理，从而导致 UI 界面冻结。最终，操作系统会终止这种行为有问题的应用程序。
- 更新 UI 的操作（比如阻塞式网络调用完成的时候）必须在主线程中运行。我们必须以某种方式请求操作系统在主线程中调用 UI 更新的代码。

令人兴奋的是，RxJava 对此有两种内置的机制。我们可以使用 subscribeOn() 在后台运行一些具有副作用的任务，并且能够很容易地通过 observeOn() 跳回主线程。4.9.2 节对这两个操作符进行了详细介绍，它们非常适合 Android 的使用场景。我们需要的仅仅是一个特殊的 Scheduler，它能够感知 Android 环境及其主线程。这个 Scheduler 已经在 4.9.1 节中实现了，幸而，这里并不需要自行实现这个 Scheduler。要开始在 Android 中使用 RxJava 的旅程，你需要添加下面这个小的依赖。

```
compile 'io.reactivex:rxandroid:1.1.0'
```

这个小的库会将 AndroidSchedulers 类添加到 CLASSPATH 中。如果想要在 Android 中使用 RxJava 编写并发代码，这是必备的。我们还是举例来阐述 AndroidSchedulers 的用法。在下面的例子中，我们将会调用 Meetup API（参见 8.1.2 节），获取指定位置附近的城市列表并将它们展现出来。

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        meetup
            .listCities(52.229841, 21.011736)
            .concatMapIterable(extractCities())
            .map(toCityName())
            .toList()
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
```

```
            .subscribe(
                    putOnListView(),
                    displayError());
    }

    //...

});
```

本章是全书中唯一没有使用 Java 8 中的 lambda 表达式的地方。因为撰写本章的时候，Android 只支持 Java 7，不允许使用原生的闭包。[1] 因此，我们将匿名内部类抽取到了单独的方法中，以提升可读性。如果你觉得这种方式过于烦琐（即便是不使用 RxJava 的场景），那么可以尝试体验一下 retrolambda，它能够将 lambda 表达式向后迁移至旧版本的 Java，并且能够在 Android 上运行。在普通的 Android 中，所有的转换和回调将会如下所示。

```
//Cities::getResults
Func1<Cities, Iterable<City>> extractCities() {
    return new Func1<Cities, Iterable<City>>() {
        @Override
        public Iterable<City> call(Cities cities) {
            return cities.getResults();
        }
    };
}

//City::getCity
Func1<City, String> toCityName() {
    return new Func1<City, String>() {
        @Override
        public String call(City city) {
            return city.getCity();
        }
    };
}

//cities -> listView.setAdapter(...)
Action1<List<String>> putOnListView() {
    return new Action1<List<String>>() {
        @Override
        public void call(List<String> cities) {
            listView.setAdapter(new ArrayAdapter(
    MainActivity.this, R.layout.list, cities));
        }
    };
}

//throwable -> {...}
Action1<Throwable> displayError() {
    return new Action1<Throwable>() {
        @Override
```

注 1：这会随着 Android N 的发布而改变。更多相关信息，请参阅使用 Java 8 的功能指南。

```
            public void call(Throwable throwable) {
                Log.e(TAG, "Error", throwable);
                Toast.makeText(MainActivity.this,
                               "Unable to load cities",
                               Toast.LENGTH_SHORT)
                         .show();
            }
        };
    }
```

接下来描述一下都发生了什么。点击按钮的时候（参见 8.1.4 节），样例会通过 Retrofit 发起一个 HTTP 请求。Retrofit 会生成一个 `Observable<Cities>`，随后只抽取相关的信息对其进行进一步的转换。最终，会得到代表附近城市的 `List<String>`。这个列表最终会在界面上展现出来。

这两个调度器的使用至关重要。如果没有 `subscribeOn()`，Retrofit 将会使用调用者的线程来发起 HTTP 调用，这会导致 `Observable` 阻塞。这意味着，HTTP 请求将会试图阻塞 Android 主线程，它会立即被操作系统执行，并且会由于出现 `NetworkOnMainThread-Exception` 而失败。按照传统方式在后台运行网络代码，要么创建新的 `Thread`，要么使用 `AsyncTask`。`subscribeOn()` 的优势非常明显：代码更加简洁，侵入性更低，而且通过 `onError` 通知提供了内置的声明式错误处理。

对 `observeOn()` 的调用同等重要。所有的转换完成之后，我们仅在主线程中调用 UI 更新，因为希望主线程上的处理越少越好。如果没有 `observeOn()` 将执行切换到 `mainThread()`，那么 `Observable` 会尝试更新后台线程中的 `listView`，而这会由于出现 `CalledFromWrongThreadException` 而立即失败。同样，`observeOn()` 要比 `android.os.Handler` 类（`AndroidSchedulers.mainThread()` 底层用到了这种方式）中的 `postDelayed()` 更加便利。

调度器的灵活性和 API 的简洁性对于 Android 开发人员非常有吸引力。RxJava 提供了一种更简单、整洁和安全的方式来处理移动设备上并发编程的复杂性。

关于内存泄漏

上面的样例有一个主要的缺陷，它可能会导致内存泄漏。`Observer` 持有了对这个封闭的 Android `Activity` 的引用，而且会存活得更久。这个问题已经在 8.1.1 节进行了阐述和解决。

8.1.4　将UI事件作为流

从语义层面上讲，RxJava 的目标是避免出现**回调地狱**，这是通过将内嵌回调替换为声明式转换实现的。因此，将 `Observable` 封闭到 `setOnClickListener()` 中看上去并不能让人满意。幸好，有一个库能够将 Android UI 事件转换为流。只需要将如下依赖添加到项目中。

```
compile 'com.jakewharton.rxbinding:rxbinding:0.4.0'
```

从现在开始，就可以将命令式的回调注册替换为便利的管道。

```
RxView
        .clicks(button)
        .flatMap(listCities(52.229841, 21.011736))
        .delay(2, TimeUnit.SECONDS)
        .concatMapIterable(extractCities())
        .map(toCityName())
        .toList()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
                putOnListView(),
                displayError());

Func1<Void, Observable<Cities>> listCities(final double lat, final double lon) {
    return new Func1<Void, Observable<Cities>>() {
        @Override
        public Observable<Cities> call(Void aVoid) {
            return meetup.listCities(lat, lon);
        }
    };
}
```

现在，不再通过注册回调在本地创建和转换 `Observable`，而是首先使用 `Observable<Void>` 代表单击按钮。单击按钮不会传递任何信息，所以它是 `Void`。每个单击事件会触发一个异步的 HTTP 请求，该请求会返回 `Observable<Cities>`。其他的内容完全相同。如果你认为这样做只是提升了可读性，那么考虑一下组合多个 GUI 事件流的场景。

假设有两个文本域，其中一个用来输入纬度，另一个用来输入经度。它们中的任何一个发生变化的时候，都会发起一个 HTTP 请求，查看该位置附近所有的城市。但是，为了避免用户输入期间产生不必要的网络流量，我们想要实现一个特定的延迟。只有 1 秒内文本域的值没有变化的时候，才会初始化网络请求。这非常类似于自动补全的文本域，它有一个轻微的延迟，避免产生大量的网络使用。但是本例必须要同时考虑两个输入域。使用 RxJava 和 RxBinding 的实现非常优雅。

```
import android.widget.EditText;
import com.jakewharton.rxbinding.widget.RxTextView;
import com.jakewharton.rxbinding.widget.TextViewAfterTextChangeEvent;


EditText latText = //...
EditText lonText = //...

Observable<Double> latChanges = RxTextView
    .afterTextChangeEvents(latText)
    .flatMap(toDouble());
Observable<Double> lonChanges = RxTextView
    .afterTextChangeEvents(lonText)
    .flatMap(toDouble());

Observable<Cities> cities = Observable
    .combineLatest(latChanges, lonChanges, toPair())
    .debounce(1, TimeUnit.SECONDS)
    .flatMap(listCitiesNear());
```

所有转换如下所示（请注意，在不能使用 lambda 的时候，代码会变得非常烦琐）。

```
Func1<TextViewAfterTextChangeEvent, Observable<Double>> toDouble() {
    return new Func1<TextViewAfterTextChangeEvent, Observable<Double>>() {
        @Override
        public Observable<Double> call(TextViewAfterTextChangeEvent e) {
            String s = e.editable().toString();
            try {
                return Observable.just(Double.parseDouble(s));
            } catch (NumberFormatException e) {
                return Observable.empty();
            }
        }
    };
}

//return Pair::new
Func2<Double, Double, Pair<Double, Double>> toPair() {
    return new Func2<Double, Double, Pair<Double, Double>>() {
        @Override
        public Pair<Double, Double> call(Double lat, Double lon) {
            return new Pair<>(lat, lon);
        }
    };
}

//return latLon -> meetup.listCities(latLon.first, latLon.second)
Func1<Pair<Double, Double>, Observable<Cities>> listCitiesNear() {
    return new Func1<Pair<Double, Double>, Observable<Cities>>() {
        @Override
        public Observable<Cities> call(Pair<Double, Double> latLon) {
            return meetup.listCities(latLon.first, latLon.second);
        }
    };
}
```

首先，每次内容发生变化的时候，`RxTextView.afterTextChangeEvents()` 就会转换 `EditText` 调用的声明式回调。分别为经度和纬度创建两个这样的流。在运行时，样例会将 `TextViewAfterTextChangeEvent` 转换为 `double` 类型，此时会丢弃不合法的输入。有了两个 `double` 类型的流之后，使用 `combineLatest()` 将它们联合起来，每当任何一个输入发生变化，样例就会接收成对的流。最后一块内容是 `debounce()`（参见 6.1.4 节），在形成这样一个配对信息之前，它会等待 1 秒，防止随后立即对文本域进行编辑（同时适用于经度和纬度）。借助 `debounce()`，在用户输入的时候，能够避免不必要的网络调用。应用程序的其他部分和以前一样。

这个示例很好地阐述了反应式编程是如何从 Retrofit 传播到用户组件的，这样，整个应用程序就变成了流的组合。不过要确保最后取消了对 `afterTextChangeEvents()` 的订阅，否则会导致内存泄漏。

8.2　使用Hystrix管理失败

> 分布式系统有这样的特点：一台你甚至不知道它存在的计算机，如果它出现了故障，有可能会导致你自己的计算机无法使用。
>
> ——Leslie Lamport，1987

RxJava 有很多操作符，它们支持编写可扩展、反应式和有弹性的应用程序。

- 通过 `Scheduler` 实现声明式并发（参见 4.9 节）。
- 超时（参见 7.1.3 节）和各种错误处理机制（参见 7.1 节和 7.1.4 节）。
- 使用 `flatMap()` 实现并行处理（参见 4.6 节），同时限制并发的数量（参见 3.1.5 节）。

不过，为了编写健壮和有弹性的应用程序，尤其是在云环境或使用微服务架构的情况下，我们需要更多非 RxJava 核心具备的特性。本节将会简要介绍一下 Hystrix，这是一个在分布式环境中管理、隔离和处理失败的库。Hystrix 允许包装可能会出现失败的行为，并围绕这些代码采取非常智能的逻辑。包括以下几点。

- 舱壁模式（bulkhead pattern），即在一定时间内完全切断不正常的行为。
- 快速失败，通过超时、限制并发、实现**断路器**（circuit breaker）等方式实现。
- 批量请求，通过将多个小的请求合并为一个大请求实现。
- 收集、发布和可视化性能统计信息。

Hystrix 最强大的功能之一就是断路器，也就是暂时关闭故障依赖的一种机制，这样失败就不会级联。如果在分布式系统中，失败没有进行恰当地处理，那么它们很容易传递到下游的依赖中，就像异常会在栈中传递一样。在分布式系统中，一个终端用户的请求可以轻易地向上游依赖发送数十个甚至上百个请求。一个出现故障的服务，即便是非必要的服务，都可能会导致整个系统的崩溃，让每个请求都失败。

有意思的是，响应慢的服务要比失败的服务更加糟糕。如果用户的请求立即失败，并且给出友好的错误信息，这种情况是糟糕的。但是，如果用户根本没有得到任何的响应，而只能无限地等待下去，那么情况就更糟糕了。遇到这种情况，用户常见的反应就是尝试刷新网页。大多数时候，这其实并没有什么助益，只会启动另外一个请求，进一步加重系统的负担。一个慢服务会导致进一步的级联，从而让整个系统陷入停顿。使用慢服务的其他所有服务会突然变得非常缓慢，而且这种情况会递归级联。Hystrix 试图屏蔽这种破坏性的依赖关系并停止故障的级联。

8.2.1　使用Hystrix的第一步

本书介绍 Hystrix 有多重原因。首先，它是基于 RxJava 构建的，这样，它就成为了现实生活中 Reactive Extensions 的一个绝佳的实际样例。其次，我们可以调用 Hystrix 命令并得到 `Observable` 类型的返回值。最后，也是最重要的，Hystrix 支持非阻塞的命令（参见 8.2.2 节）。

但是在继续讲解之前，先看一下如何将 Hystrix 用到最简单的阻塞式场景中。根据经验，要将 Hystrix 用在执行脱离了进程或机器的地方。发起网络调用（也包括访问 I/O）会显著增加失败的风险：风险可能包括难以预料的延迟、网络分区或数据包丢失。在识别出这样

有潜在风险的代码块之后，使用 HystrixCommand 对其进行包装。

```
import org.apache.commons.io.IOUtils;
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;

class BlockingCmd extends HystrixCommand<String> {

    public BlockingCmd() {
        super(HystrixCommandGroupKey.Factory.asKey("SomeGroup"));
    }

    @Override
    protected String run() throws IOException {
        final URL url = new URL("http://www.example.com");
        try (InputStream input = url.openStream()) {
            return IOUtils.toString(input, StandardCharsets.UTF_8);
        }
    }

}
```

样例将可能会失败的阻塞式代码封装到一个 run() 方法中。这个方法的 T 类型是通过 HystrixCommand<T> 泛型定义的。如果想要参数化操作（比如使用不同 URL），那么它必须要通过构造器传递进来。HystrixCommand 是**命令**（Command）设计模式的一种实现，这个模式是在 Erich Gamma 等人编写的经典《设计模式：可复用面向对象软件的基础》中定义的。

有了 HystrixCommand 实例之后，必须要以某种方式执行它。这里有两种执行方式：按照命令的方式执行，或者获取先于 Java 8 提供的一个 Future 实例。不过这两种方式都没什么吸引力。

```
String string         = new BlockingCmd().execute();
Future<String> future = new BlockingCmd().queue();
```

execute() 会通过一个安全网，间接调用 run() 方法，这个安全网包括了超时、高级的错误处理等，8.2.3 节将介绍更多相关内容。这个方法会一直阻塞，只有底层的 run() 完成或抛出异常的时候，它才会返回。在这种情况下，异常会传递给调用者。而 queue() 是非阻塞的，但是它会返回 Future<T>。旧的 Future 接口并不具备真正的反应性，因此本书不会过多关注 execute() 和 queue()。

> 注意，在这里每次都会创建新的 BlockingCmd，我们不能跨多次执行重用命令实例。HystrixCommand 命令应该是在执行之前直接创建的，不能进行重用。在实践中，通常会在创建时对命令进行参数化，这样，实例的可重用性也值得怀疑了。

Hystrix 支持将 Observable 作为一等公民，[2] 可以将命令的结果返回为流。

注 2：事实上，execute() 是通过 queue() 实现的，而 queue() 又是通过 toObservable() 实现的。

```
Observable<String> eager = new BlockingCmd().observe();
Observable<String> lazy  = new BlockingCmd().toObservable();
```

observe() 和 toObservable() 在语义上的差异非常重要。toObservable() 会将一个命令转换成延迟执行的 cold 类型的 Observable，换言之，在实际有人订阅该 Observable 之前不会执行这个命令。另外，这个 Observable 不会进行缓存，即每次 subscribe() 都会触发命令的执行。与之不同，observe() 会以异步的方式直接执行该命令，返回一个 hot 类型且支持缓存的 Observable。正如 4.3 节介绍的，延迟执行的 Observable 非常便利，比如可以在任何时间点创建它们，但是不进行订阅。这样，就能避免产生网络调用这样的副作用，还可以非常有效地进行批量请求。但是，cold 类型的 Observable 也有一个风险：如果有多个订阅者，那么将会被调用多次。在这种情况下，cache() 操作符就能发挥作用了。延迟执行一般能够实现最高效率的并发，所以应该优先使用 toObservable()，而不是 observe()，除非有充分的理由立即调用某条命令。

有了 Observable 之后，我们就可以使用各种操作符了，例如对失败的命令使用 retry()。

```
Observable<String> retried = new BlockingCmd()
    .toObservable()
    .doOnError(ex -> log.warn("Error ", ex))
    .retryWhen(ex -> ex.delay(500, MILLISECONDS))
    .timeout(3, SECONDS);
```

上述管道会调用一条命令，如果失败，样例会在 500 毫秒之后重试。但是，重试可能会耗费 3 秒的时间，如果超出了这个时间，就会抛出 TimeoutException（参见 7.1.3 节）。接下来将介绍 Hystrix 内置的超时功能如何发挥作用。

8.2.2　使用HystrixObservableCommand的非阻塞命令

如果在设计应用程序的时候就使用到了 RxJava，那么很可能已经对第三方服务或未知库的调用建模为 Observable。基本的 HystrixCommand 只支持阻塞式的代码。但是，如果与外部世界的交互已经是 Observable，而且还想使用 Hystrix 实现进一步的保护，那么 HystrixObservableCommand 就是合适的选择。

```
public class CitiesCmd extends HystrixObservableCommand<Cities> {

    private final MeetupApi api;
    private final double lat;
    private final double lon;

    protected CitiesCmd(MeetupApi api, double lat, double lon) {
        super(HystrixCommandGroupKey.Factory.asKey("Meetup"));
        this.api = api;
        this.lat = lat;
        this.lon = lon;
    }

    @Override
    protected Observable<Cities> construct() {
        return api.listCities(lat, lon);
    }
}
```

MeetupApi 在 8.1.2 节中介绍过，它可以返回 Observable<Cities>。Hystrix 透明地包装了这个 Observable，从而添加了容错处理的特性，稍后会对其进行介绍。相对于 BlockingCmd，CitiesCmd 更加实用，因为在它的构造器中接收了一些参数。在单元测试中，可以传入一个 MeetupApi 的存根实例，以验证命令行为。

与 HystrixCommand 相比，HystrixObservableCommand 的优势在于其操作不需要线程池。HystrixCommand 始终在一个有界的线程池中执行，而 Observable 命令不需要任何额外的线程。当然，construct()（注意这里不再是 run() 了）返回的 Observable 依然可以使用某些依赖于底层实现的线程。

现在，知道了如何在 Hystrix 中创建命令，以及它们如何适应 RxJava 生态系统，接下来介绍一下 Hystrix 都提供了哪些实际的功能。

8.2.3 舱壁模式和快速失败

舱壁（bulkhead）是横跨轮船船体的大墙，它们会形成水密舱。如果出现漏水，舱壁会将水隔离在一个水密舱中，防止轮船沉没。相同的工程理念可以应用到分布式系统。如果应用程序中的一个组件出现失败，那么它应该被隔离起来。这样即便某个组件出现了故障，整个系统也能正常运行。

另外一个可以很好地应用于软件领域的工程模式是**断路器**（circuit breaker）。断路器的责任是中断电流，避免各种设备过载甚至着火。在危险解除之后，断路器可以复位（以人工或自动化的方式）。但是，这是否会导致电灯、加热器或者（最糟糕的情况下）路由器出现断点呢？不一定。其他电路网络可能会收到其他断路器保护，因此它们依然能够运行。最重要的是，你的房子没有着火。

Hystrix 在系统集成领域实现了这两种模式。每条命令都有超时时间（默认为 1 秒）和并发限制（默认情况下，给定组只能有 10 条并发命令）。这些严格的限制确保了命令不会消耗太多的资源，比如线程和内存。同时，使用超时功能能够确保不会引入过多的延迟。我们可以将这种行为与轮船上的舱壁进行对比，如果某一个依赖开始出现失败（注意，过高的延迟和失败几乎无法区分），这个问题将不再影响整个系统。超时和有限的并发能够明显减少阻塞在外部系统的线程数量。

断路器则更加聪明。假设，某个以前 100 毫秒就能响应的依赖，现在几乎要等待 1 秒才会产生超时。如果对这个行为不正常的依赖的调用是整个处理流程的一部分，那么现在几乎每个事务都会比原来慢一秒。如果没有 Hystrix，这个延迟可能会大得更多，不过使用纯粹的 RxJava 也可以实现超时功能。Hystrix 能做的并不局限于此。如果它发现特定的命令在给定的时间窗口（默认为 10 秒）频繁地（默认为全部调用的 50%）失败（出现异常或超时），那么它就会打开一个回路。接下来发生的事情就会很有意思了，Hystrix 将不再调用出现故障的命令。相反，它会立即抛出一个异常，快速失败。

接下来实际看看 Hystrix。首先，使用 Mockito 模拟（mock-up）MeetupApi，这样，它会始终由于一些不可接受的延迟而失败，如下所示。

```
import static org.mockito.BDDMockito.given;
import static org.mockito.Matchers.anyDouble;
```

```
import static org.mockito.Mockito.mock;


MeetupApi api = mock(MeetupApi.class);
given(api.listCities(anyDouble(), anyDouble())).willReturn(
    Observable
        .<Cities>error(new RuntimeException("Broken"))
        .doOnSubscribe(() -> log.debug("Invoking"))
        .delay(2, SECONDS)
);
```

默认的超时时间是 1 秒，所以实际上我们永远都不会看到“Broken”异常，因为超时会首先生效。现在，我们想要多次并发调用 MeetupApi 并查看 Hystrix 的行为。

```
Observable
    .interval(50, MILLISECONDS)
    .doOnNext(x -> log.debug("Requesting"))
    .flatMap(x ->
            new CitiesCmd(api, 52.229841, 21.011736)
                .toObservable()
                .onErrorResumeNext(ex -> Observable.empty()),
        5)
```

使用 interval() 操作符，每 50 毫秒发布一个事件。每当遇到这种事件，样例都会触发 CitiesCmd 命令并将异常隐藏掉。注意，在实际的项目中，我们至少应该使用 doOnError() 回调来记录异常信息。Hystrix 每隔 50 毫秒就会调用命令，并在 1 秒之后得到超时的通知。这个命令实际执行得更慢，但是 Hystrix 会提前中断它。我们进行订阅并运行这个程序的时候，会发现 CitiesCmd 被调用几次，但随后就突然停止。尽管“Requesting”消息依然每隔 50 毫秒出现，但是命令不会再被调用了。

通过一些启发式的算法，Hystrix 判断 CitiesCmd 已经出现了问题，所以不再调用它。相反，当我们尝试调用这个命令，产生的 Observable 会立即失败并伴有一个异常。这里，断路器介入并导致对命令的调用快速失败。命令根本就不会被调用，因为 Hystrix 认为它会一直失败，没有调用的必要。失败率超过 50% 的时候，断路器就会打开，调用命令的后续尝试都会瞬间失败。在失败时，Hystrix 会假定出现异常或超时。

断路器有双重优点。从调用命令的应用程序的角度，不管怎样它最终都会失败，但是能够更快地得到响应，这会带来更好的用户体验。更有意思的是在服务器端，或者是说命令中请求要访问的目标。如果命令持续失败或者超时，那么这可能是依赖（另外的服务、数据库）遇到困难的一个信号。这可能是重启、流量峰值或者长时间的 GC 停顿导致的。我们通过断路器切断这个命令，从而给了系统喘息的空间。负载峰值结束或内部任务队列清空时，系统可能会重新恢复响应并健康状态。这样，能够防止系统对自己发起分布式拒绝服务攻击（Distributed Denial of Service，DDoS）。

那么，Hystrix 是怎样识别出下游依赖重新恢复健康状态并将断路器关闭的呢？幸而，这个过程是自动完成的。在前面的样例中，我们在某个时间点看不到 Invoking 日志信息了。这意味着断路器已经打开，命令不再执行了。但事实并非完全如此。每隔一段时间（默认是每 5 秒），Hystrix 就会允许一个请求通过并执行命令，以此来检查此时是否已经恢复健康状态。与此同时，其他客户端依然会快速失败。如果这个请求成功，Hystrix 就会假定命令

已经恢复健康状态，并将断路器关闭；否则，断路器会依然处于打开状态。

这种特性叫作**自愈**（self-healing），在计算机系统中是一个重要的概念。Hystrix 能够在两个方面提供帮助。通过临时关闭出现问题的命令，让下游的依赖恢复健康。在它们恢复之后，系统将回归正常的操作流程。如果没有这种机制，即便一个很小的故障都可能导致级联失败，为了恢复各个组件的稳定性，我们可能还需要手动重启。

8.2.4 批处理和合并命令

Hystrix 最高级的特性之一就是请求的批量处理。假设处理一个上游请求的过程中，需要向下游发起多次小规模的请求。例如，我们想要展现一个图书的列表，每本书都需要请求一个外部系统来获取它的评分。

```
Observable<Book> allBooks() { /* ... */ }
Observable<Rating> fetchRating(Book book) { /* ... */ }
```

`allBooks()` 方法会返回我们想要处理的 `Book` 流，`fetchRating()` 则会为每本书 `Book` 返回一个评分 `Rating`。原始实现可能会遍历所有的图书并依次检索其 `Rating`。幸而，使用 RxJava 异步运行子任务非常容易。

```
Observable<Rating> ratings = allBooks()
        .flatMap(this::fetchRating);
```

图 8-1 和图 8-2 对比了序列化调用 `fetchRatings()` 与使用 `flatMap()` 的差异。在这几个阶段中，`send` 指的是传输请求，`proc` 指的是服务器端处理，而 `recv` 指的是传输响应。图 8-1 展现了序列化获取数据的过程。

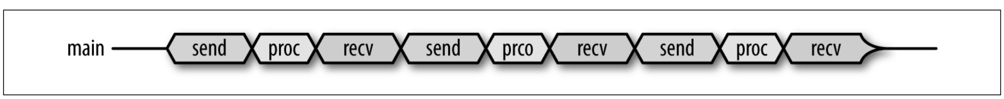


图 8-1

图 8-2 展示了使用 `flatMap()` 获取数据的场景。

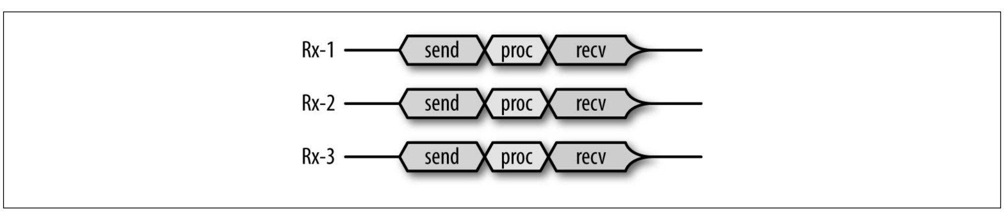


图 8-2

这运行得非常好，通常也能看到令人满意的性能。所有的 `fetchRatings()` 都是并发执行的，从而极大地改善了延迟。但是，如果每次调用 `fetchRatings()` 都会带来固定量的网络延迟，那么为几十本图书而调用它就非常浪费了。如果为所有图书只发送一个批量请求，并在一个响应中得到所有图书的评分信息，那么以下这种方式看上去会更有效，如图 8-3 所示。



图 8-3

注意，这种情况下所有的阶段（发送、处理和接收）在一定程度上都会稍慢一些。所有这些阶段都要传输或处理更多的数据，这是可以理解的。因此，相对于发送多个小规模的请求，总的延迟实际上会更高。这种改善值得怀疑，但是我们必须从更高的角度来看问题。

尽管单个请求的延迟会增加，但是系统的吞吐量可能会得到大幅提升。我们能够处理的并发连接数、网络吞吐以及 JVM 线程都是有限且稀缺的资源。如果请求的依赖吞吐量有限，那么使用并发来实现的几个事务就有可能使其饱和。私自使用 `flatMap()` 能够改善单个请求的延迟，但是这样会使资源饱和，从而降低其他请求的性能。因此，我们希望牺牲一点延迟，避免对下游依赖产生太大的负载，以获取更好的吞吐量。最终，延迟实际上也能得到改善：请求在共享资源方面更加公平，因此延迟也更具可预测性。

那么，该如何实现批处理呢？ Hystrix 能够感知执行的每个命令。要同时执行两条类似的命令（比如获取两个 `Rating`）时，它能够将这两条命令合并为一个更大的批处理命令。这个批处理命令会被调用，而且批量响应到达时，这些响应数据会映射回原始的单个请求。首先，我们需要一个批处理命令的实现，它能够一次性检索多个 `Rating` 信息，如下所示。

```
class FetchManyRatings extends HystrixObservableCommand<Rating> {

    private final Collection<Book> books;

    protected FetchManyRatings(Collection<Book> books) {
        super(HystrixCommandGroupKey.Factory.asKey("Books"));
        this.books = books;
    }

    @Override
    protected Observable<Rating> construct() {
        return fetchManyRatings(books);
    }

}
```

`fetchManyRatings()` 方法接收 `books` 作为参数，并且发布多个 `Rating` 实例。在内部，它可以通过一条 HTTP 请求，获取多个评分信息，这与 `fetchRating(book)` 截然不同，后者永远只能获取一个评分信息。请求多个 `Rating` 显然会更慢，但是肯定比序列化获取 `Rating` 快一些。但是，我们并不想手动管理批处理中的多个请求，然后再拆解批处理的响应。如果只处理一个事务，这可能还简单一些；但是如果有多个并发客户端，每个客户端都请求 `Rating`，又该如何处理呢？来自两个浏览器的独立请求访问服务器的时候，我们依然想对这两个请求进行批处理，并且只对下游发送一次调用。这就需要线程间的同步，以及所有请求的全局注册表。假设某个线程试图调用给定的命令，而另外一个线程在几毫秒之后调用了相同的命令（参数不同）。我们想要在第一个请求尝试启动命令之后等待一会儿，以防稍后有另外的线程尝试调用相同的命令。如果是这样，我们将捕获这两个请求，把它们

合并在一起，只发送一次批处理请求，然后将批处理响应映射回各个独立请求。这正是 Hystrix 在样例的帮助下做的事情。

```
public class FetchRatingsCollapser
    extends HystrixObservableCollapser<Book, Rating,
Rating, Book> {

    private final Book book;

    public FetchRatingsCollapser(Book book) {
        //后面进行阐述
    }

    public Book getRequestArgument() {
        return book;
    }

    protected HystrixObservableCommand<Rating> createCommand(
        Collection<HystrixCollapser.CollapsedRequest<Rating, Book>> requests) {
        //后面进行阐述
    }

protected void onMissingResponse(
        HystrixCollapser.CollapsedRequest<Rating, Book> r)
{
      r.setException(new RuntimeException("Not found for: "
+ r.getArgument()));
}

    protected Func1<Book, Book> getRequestArgumentKeySelector() {
        return x -> x;
    }

    protected Func1<Rating, Rating> getBatchReturnTypeToResponseTypeMapper() {
        return x -> x;
    }

    protected Func1<Rating, Book> getBatchReturnTypeKeySelector() {
        return Rating::getBook;
    }

}
```

这里有很多代码，接下来逐步进行分析。我们想要检索指定 `Book` 的 `Rating`，于是创建了一个 `FetchRatingsCollapser` 实例，如下所示。

```
Observable<Rating> ratingObservable =
    new FetchRatingsCollapser(book).toObservable();
```

得益于 `HystrixObservableCollapser`，客户端代码完全不知道正在发生的批处理和命令合并。从外部看，使用方式仿佛依然是为一个 `Book` 获取 `Rating`。但是在内部，一些比较有意思的细节使得我们可以进行批处理。首先，在构造器中，除了存储该请求的 `Book`，还配置了请求的合并。

```
public FetchRatingsCollapser(Book book) {
    super(withCollapserKey(HystrixCollapserKey.Factory.asKey("Books"))
            .andCollapserPropertiesDefaults(HystrixCollapserProperties.Setter()
                    .withTimerDelayInMilliseconds(20)
                    .withMaxRequestsInBatch(50)
            )
            .andScope(Scope.GLOBAL));
    this.book = book;
}
```

withTimerDelayInMilliseconds() 配置的 20 毫秒是合并发生的时间窗口的长度（默认为 10 毫秒）。第一个请求发生后，在它真正被调用之前会经过 20 毫秒的延迟。这段时间内，Hystrix 会等待其他请求抵达，这些请求可能由其他线程发起。Hystrix 会试探性地延迟第一个请求，以查看是否有同种类型的命令抵达。当时间耗尽或者排队的请求已经达到了 50 个（withMaxRequestsInBatch(50) 参数设置的），大门将会关闭。此时，假定 Hystrix 将在一个批次中调用所有排队的命令。但是，Hystrix 并不会神奇地将命令批量处理为一个，我们必须指导它实现这一点。以下代码段展示了这个功能的实现。

```
protected HystrixObservableCommand<Rating>
createCommand(
    Collection<HystrixCollapser.CollapsedRequest<Rating,
Book>> requests) {
    List<Book> books = requests.stream()
            .map(c -> c.getArgument())
            .collect(toList());
    return new FetchManyRatings(books);
}
```

createCommand() 方法负责将多个单独的请求转换为一个批处理命令。它会接收 20 毫秒时间段内收集到的所有请求，它们现在应该合并为单个、批处理的命令。在这个场景中，构建一个 FetchManyRatings 命令的实例，它会为所有 Book 获取 Rating 数据。Hystrix 随后会触发批处理命令并订阅所有响应。注意，HystrixObservableCommand 允许返回多个值，而这就是我们想要实现的功能。

FetchManyRatings 开始出现值的时候，必须要以某种方式将 Rating 实例与独立的请求匹配起来。此时需要注意，可能会有多个独立的线程和事务，每个线程和事务都在等待一个 Rating 结果。批处理的响应路由和分发至单独的请求，会通过如下的方法或多或少地自动完成。

- getRequestArgumentKeySelector()

  这个方法会将单个请求的参数（Book）映射为一个 key，随后使用这个 key 映射批处理的响应。这个场景只是使用了相同的 Book 实例，因此会进行 x -> x 恒等转换。

- getBatchReturnTypeToResponseTypeMapper()

  这个方法会将批处理响应的每个条目映射为单个响应。同样，在这个场景中，使用 x -> x 恒等转换就足够了。

- getBatchReturnTypeKeySelector()

  这个方法可以让 Hystrix 分辨特定的响应（Rating）是应对哪个请求 key（Book）的。简单起见，批处理响应返回的每个 Rating 都有一个 getBook() 方法，用来指定与它相关的 Book。

这些方法（尤其是最后一个，`getBatchReturnTypeKeySelector()`）就绪之后，Hystrix 会根据请求的 key（`Book`）准备一个映射，一旦新的 `Rating` 从批处理响应中出现，它就能自动地将响应与请求进行匹配。

让批处理开始运行，我们需要进行很多准备工作，但是很快就能得到回报。多个客户端访问相同的下游依赖（例如，缓存服务器）时，可以将多个请求收集为一个。这会明显减少带宽的成本。当依赖成为瓶颈，而吞吐量又有限的时候，合并请求将极大地减少依赖的负载。但是，批处理在客户端引入了额外的延迟。按照默认配置 10 毫秒（`withTimerDelayInMilliseconds(10)`），在高负载的情况下，每个请求平均会延迟 5 毫秒。实际的延迟取决于当前的请求是刚刚启动一个计时器，还是出现在这个批次即将结束之前。

需要注意，在低负载的时候，批处理并没有太大的价值。如果在每个批次中，请求的数量很少超过一个，那么只是给每个请求都添加了额外的 10 毫秒延迟而已。这是 Hystrix 毫无意义地等待其他请求的时间。因此，对批次的调优是非常重要的。如果你的定时器延迟是 10 毫秒，那么只有当每秒至少有 1000 个请求的时候，批处理才有意义。否则，很少会出现多个请求形成一个批处理的情况。

调优 `withTimerDelayInMilliseconds`

为了将更多的请求放到一个批次中，我们会忍不住设置较长的延迟。像 100 毫秒甚至 1 秒这样的值都是可以的，但是这种情况在离线系统中效果会最好，因为这会产生大量的流量，而且延迟并不是什么问题。

在高负载的情况下，批处理是运行得最好的特性。因此，Hystrix 提供了非常全面的监控机制，帮助我们理解整体的系统性能。

8.2.5 监控和仪表盘

为了正常运行，随着时间的推移，Hystrix 必须在内部为每个命令收集大量的统计数据，比如统计成功和失败调用的数量以及响应时间的分布。如果这些宝贵的数据只能在这个库中使用，就显得有些自私了。但是不必担心：Hystrix 提供了多种方式来使用这些数据。我们可以订阅 Hystrix 提供的各种类型的流，这些流会发布 Hystrix 库中发生的事件。例如，以下代码创建了一个 `HystrixCommandCompletion` 事件流，每当 `FetchRating` 命令完成时，这个流都会发布事件。

```
import com.netflix.hystrix.metric.HystrixCommandCompletion;
import com.netflix.hystrix.metric.HystrixCommandCompletionStream;


Observable<HystrixCommandCompletion> stats =
    HystrixCommandCompletionStream
    .getInstance(HystrixCommandKey.Factory.asKey("FetchRating"))
    .observe();
```

`HystrixCommandCompletionStream` 是这种流的工厂，但是还有很多其他的流，比如 `HystrixCommandStartStream` 和 `HystrixCollapserEventStream`。将这些流嵌入到应用程序之

后，就能构建更加复杂的监控系统了。例如，我们想要知道给定的命令每秒失败的次数，那么可以尝试如下的代码。

```
import static com.netflix.hystrix.HystrixEventType.FAILURE;


HystrixCommandCompletionStream
        .getInstance(HystrixCommandKey.Factory.asKey("FetchRating"))
        .observe()
        .filter(e -> e.getEventCounts().getCount(FAILURE) > 0)
        .window(1, TimeUnit.SECONDS)
        .flatMap(Observable::count)
        .subscribe(x -> log.info("{} failures/s", x));
```

但是，基于这些流构建监控基础设施还需要一些设计和相关工作。同时，我们可能还希望将监控功能从实际的应用中抽取出来。Hystrix 借助 `hystrix-metrics-event-stream` 模块，能够将所有聚合的指标通过 HTTP 推送出来。如果应用程序已经运行或者使用嵌入式的 Servlet 容器，那么只需要给映射添加一个内置的 `HystrixMetricsStreamServlet`。否则，我们需要自行启动一个小型的容器。

```
import
com.netflix.hystrix.contrib.metrics.eventstream.
    HystrixMetricsStreamServlet;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.servlet.ServletContextHandler;
import org.eclipse.jetty.servlet.ServletHolder;
import static org.eclipse.jetty.servlet.ServletContextHandler.NO_SESSIONS;

//...

ServletContextHandler context = new ServletContextHandler(NO_SESSIONS);
HystrixMetricsStreamServlet servlet = new HystrixMetricsStreamServlet();
context.addServlet(new ServletHolder(servlet), "/hystrix.stream");
Server server = new Server(8080);
server.setHandler(context);
server.start();
```

无论我们是将 Servlet 映射到已有容器，还是自行启动容器，通过上面的配置，现在就能够实时访问流动的 Hystrix 统计信息了。注意，这个连接并不是简单的请求—响应模式，而是一个服务器推送事件（Server-sent Event，SSE）流。每秒都会有一个新的统计信息包以 JSON 格式推送至客户端。

```
$ curl -v localhost:8080/hystrix.stream
> GET /hystrix.stream HTTP/1.1
...
< HTTP/1.1 200 OK
< Content-Type: text/event-stream;charset=UTF-8

ping:

data: {
    "currentConcurrentExecutionCount": 2,
    "errorCount": 0,
```

```
        "errorPercentage": 0,
        "group": "Books",
        "isCircuitBreakerOpen": false,
        "latencyExecute": {/* ... */},
        "latencyExecute_mean": 0,
        "latencyTotal": {"0":18, "25":80, "50":98, "75":120, "90":138,
                         "95":146, "99":159, "99.5":159, "100":167},
        "latencyTotal_mean": 0,
        "name": "FetchRating",
        "propertyValue_circuitBreakerErrorThresholdPercentage": 50,
        "propertyValue_circuitBreakerSleepWindowInMilliseconds": 5000,
        "propertyValue_executionIsolationSemaphoreMaxConcurrentRequests": 10,
        "propertyValue_executionTimeoutInMilliseconds": 1000,
        "requestCount": 334
        ...
    }

    data: { ...
```

即便是这个简单样例，我们也能看出它测量的是哪个命令、延迟是如何分布的（从第 0 个到第 100 个百分位）、断路器是否处于打开的状态，以及断路器的参数（错误阈值、超时等）。这种连续的数据流可以进一步被自定义监控工具或仪表盘消费。同样，Hystrix 提供了一个非常健壮的仪表盘，它几乎全部是使用 JavaScript 编写的，可以在浏览器中运行。这个独立的应用程序（在 `hystrix-dashboard` 中实现）需要的只是一个到 `hystrix.stream` 的 URL。图 8-4 展示了一个示例仪表盘。

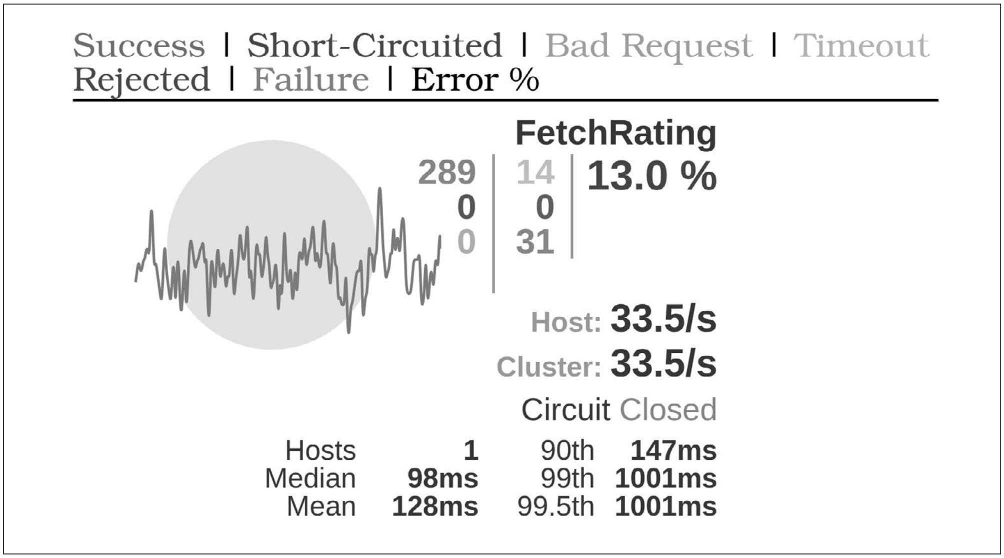


图 8-4

每条命令都有这样的一个详细图表，展示了一些重要的遥测指标细节，包括以下部分。

- 执行过的命令，按照成功的（289）、超时的（14）、失败的（31）、短路的（0）等进行分组。

- 延迟的百分位（可以看到 90% 的请求耗时不超过 147 毫秒），在图表中展现了短期的请求历史。
- 断路器状态以及整体吞吐量。
- 如果使用阻塞式 `HystrixCommand`，还会看到线程池状态。

仪表盘还能够展现通过 Turbine 聚集的其他服务器的流。这就是我们会看到主机数量和集群吞吐量的原因，当然这里的流是来自一台机器的。Hystrix 仪表盘非常有用，因为它能够以接近实时的速度快速展现多条命令的状态。它还进行了颜色的区分，如果有些命令开始失败，它们对应的图表就会变成红色。

在分布式系统中，Hystrix 是一个非常有用的工具，因为在这种环境中，失败是难以避免的。命令模式允许封装和隔离错误域。对于需要更好的错误处理的反应式应用程序，Hystrix 与 RxJava 的良好集成是一个不错的选择。

8.3　查询NoSQL数据库

目前，典型应用程序的数据高延迟主要有两个根源：网络调用（大多数是 HTTP）和数据库查询。Retrofit（参见 8.1.2 节）是一个由异步 HTTP 调用作为支撑的 `Observable` 源。在数据库访问方面，本书花费了很多的时间讨论 SQL 数据库（参见 5.3 节），因为 JDBC API 的设计，它们一直以来都是阻塞式的。在这方面，NoSQL 数据库更加现代化，通常会提供异步、非阻塞的客户端驱动。本节将会简要介绍 Couchbase 和 MongoDB 驱动，它们对 RxJava 提供了原生支持，能够为外部调用返回 `Observable`。

8.3.1　Couchbase客户端API

Couchbase 服务器是 NoSQL 家族中的一款现代文档数据库。有意思的是，在客户端 API 中，Couchbase 支持将 RxJava 作为一等公民。在与数据库进行交互的时候，Reactive Extensions 不仅仅用作包装器，还得到了 Couchbase 的官方支持和常见的用法。很多其他的存储引擎都有非阻塞、异步的 API，但是 Couchbase 选择将 RxJava 作为客户端层的最佳基础。

查询名为 `travel-sample` 的示例数据集作为样例，它有一个 ID 为 `route_14197` 的文档。在示例数据集中，路由文档如下所示。

```
{
  "id": 14197,
  "type": "route",
  "airline": "B6",
  "airlineid": "airline_3029",
  "sourceairport": "PHX",
  "destinationairport": "BOS",
  "stops": 0,
  "equipment": "320",
  "schedule": [
    {
      "day": 0,
      "utc": "22:12:00",
```

```
      "flight": "B6928"
    },
    {
      "day": 0,
      "utc": "06:40:00",
      "flight": "B6387"
    },
    ...
    {
      "day": 1,
      "utc": "08:16:00",
      "flight": "B6922"
    }
    ...
```

每次查询都会返回一个 Observable，此时，样例就可以安全地按照我们认为合适的方式转换获取的记录。

```
CouchbaseCluster cluster = CouchbaseCluster.create();
cluster
    .openBucket("travel-sample")
    .get("route_14197")
    .map(AbstractDocument::content)
    .map(json -> json.getArray("schedule"))
    .concatMapIterable(JsonArray::toList)
    .cast(Map.class)
    .filter(m -> ((Number)m.get("day")).intValue() == 0)
    .map(m -> m.get("flight").toString())
    .subscribe(flight -> System.out.println(flight));
```

AsyncBucket.get() 会返回一个 Observable<JsonDocument>。JSON 文档本质上是松散类型的，所以为了抽取有意义的信息，样例必须根据预先掌握的结构对其进行遍历。

在掌握了文档的格式之后，理解对 JsonDocument 的转换就非常容易了。转换过程首先会抽取 schedule 元素，然后过滤所有 day 为 0 的节点，并最终得到 flight 节点。Observer 最后得到的是 B6928、B6387 这样的字符串。令人惊讶的是，RxJava 同样适用于如下场景。

- 数据检索，包括超时、缓存以及错误处理。
- 数据转换，如提取、过滤、数据钻取和聚合。

这个样例展示了 Observable 抽象的强大功能，它能够将同样简洁的 API 用到各种不同的场景中。[3]

8.3.2 MongoDB客户端API

与 Couchbase 类似，MongoDB 允许存储任意的类 JSON 的文档，而无须预先定义模式。客户端库对 RxJava 提供了良好的支持，允许异步存储和查询数据。如下的样例展示了这两种功能。它首先插入 12 条文档到数据库，批量插入完成后，就会将数据再查询出来。

注 3：这个特性类似于 .NET 平台上的语言集成查询（Language Integrated Query，LINQ）。

```
import com.mongodb.rx.client.*;
import org.bson.Document;
import java.time.Month;


MongoCollection<Document> monthsColl = MongoClients
    .create()
    .getDatabase("rx")
    .getCollection("months");

Observable
    .from(Month.values())
    .map(month -> new Document()
        .append("name", month.name())
        .append("days_not_leap", month.length(false))
        .append("days_leap", month.length(true))
    )
    .toList()
    .flatMap(monthsColl::insertMany)
    .flatMap(s -> monthsColl.find().toObservable())
    .toBlocking()
    .subscribe(System.out::println);
```

`Month` 是一个 `enum` 类，具有从 `January` 到 `December` 的值，我们还可以获取闰年和非闰年任意月份的长度。首先，创建 12 个 BSON（二进制 JSON）文档，每个代表一个月份及其长度。然后，使用 `MongoCollection` 的 `insertMany()` 方法批量插入 `List<Document>`。这个操作会生成 `Observable<Success>`（这个值本身并不包含任何有意义的信息，它是单例的）。`Success` 事件出现的时候，通过调用 `find().toObservable()` 查询数据库。我们希望刚刚插入的 12 条文档都能找到。为了简洁起见，这里移除了自动分配的 `_id` 属性，最终打印的结果如下所示。

```
Document{{name=JANUARY, days_not_leap=31, days_leap=31}}
Document{{name=FEBRUARY, days_not_leap=28, days_leap=29}}
Document{{name=MARCH, days_not_leap=31, days_leap=31}}
...
```

再次强调一遍，真正的威力源于组合。借助 MongoDB 的 RxJava 驱动，我们可以很容易地同时查询多个集合，无须过多关注细节就能实现并发。如下的代码片段发起了两个对 MongoDB 的并发请求，另外还发起了一个对定价服务的请求。注意，`first()` 并不是 `Observable` 的操作符，而是 MongoDB 的操作符，它会在构造查询之后返回一个 `Observable`。`find()` 等价于 SQL 中的 `WHERE` 子句，`projection()` 代表 `SELECT.first()`，这类似于 `LIMIT 1`。

```
Observable<Integer> days = db.getCollection("months")
    .find(Filters.eq("name", APRIL.name()))
    .projection(Projections.include("days_not_leap"))
    .first()
    .map(doc -> doc.getInteger("days_not_leap"));
Observable<Instant> carManufactured = db.getCollection("cars")
    .find(Filters.eq("owner.name", "Smith"))
    .first()
```

```
        .map(doc -> doc.getDate("manufactured"))
        .map(Date::toInstant);

Observable<BigDecimal> pricePerDay = dailyPrice(LocalDateTime.now());
Observable<Insurance> insurance = Observable
    .zip(days, carManufactured, pricePerDay,
        (d, man, price) -> {
            //创建保险数据
        });
```

从技术上讲，我们可以混合任意的 Observable，而不用关心其特点和来源。上述样例发起了针对不同集合的两个 MongoDB 查询，还有另外一个 dailyPrice() 查询，它可能是 Retrofit 发起的 HTTP 调用，该查询返回的是一个 Observable。底线是：Observable 的来源无关紧要，我们可以任意地组合异步计算和请求。你是否计划将多个数据库查询与网络服务、本地文件系统操作结合在一起呢？所有这些操作都可以并发运行，并且可以同样容易地组合在一起。掌握了 RxJava 整体是如何运行的之后，就会发现每个 Observable 源在表面上都是相同的。

8.4　Camel集成

8.1.2 节介绍了如何借助 RxJava 的支持发起 HTTP 请求。但是，还有很多其他的方式进行系统集成，其中很大一部分都内置到了 Apache Camel 框架中。Camel 有一组令人赞叹的集成组件，借助它们，我们可以与 200 多个平台连接和交换抽象消息。这些平台包括 AMQP、Amazon Web Services、Cassandra、ElasticSearch、文件系统、FTP、Google API、JDBC、Kafka、MongoDB、SMTP、XMP 等。大多数组件都能推送抽象消息到客户端，举例来说，这样的消息包括收到新的 Email 或者文件系统中的新文件。

Camel 还提供了 RxJava 适配器，从而以声明式、反应式的方式使用传入的消息。

8.4.1　通过Camel来消费文件

借助 RxJava 的 Observable 和操作符，我们可以按照统一的方式集成数百种平台。例如，假设我们想要监控文件系统的新文件（与 4.8 节进行对比）。由于 Camel 对 RxJava 的支持，以下任务会非常简单。

```
CamelContext camel = new DefaultCamelContext();
ReactiveCamel reactiveCamel = new ReactiveCamel(camel);

reactiveCamel
    .toObservable("file:/home/user/tmp")
    .subscribe(e ->
        log.info("New file: {}", e));
```

这样就可以了！在创建完 DefaultCamelContext 和 ReactiveCamel 之后，就可以开始消费消息了。Camel 支持的每个集成平台都会通过 URI 的方式进行编码，在这个场景中就是 file:/home/user。使用这个 URI 来调用 toObservable() 就能得到一个通用的 Observable<Message>，每次在指定的目录中有新文件出现，它就会发布一个事件。对于每种集成平台，URI 本身会

有数十个配置项。比如，通过为 fileURI 添加 ?recursive=true&noop=true 配置，我们要求 Camel 递归查找文件，并且在发现文件后不删除它们。

8.4.2　接收来自Kafka的消息

消费轮询文件系统变化来得到数据是一种很流行的集成技术。但是，如果需要更健壮、更快速、更可靠的通信协议，那么应该选择基于 JMS 规范的消息代理（broker）或者 Kafka。Kafka 是一个开源的发布—订阅消息代理。按照设计，它默认支持容错，每秒能够处理数十万条消息。Kafka 有一个原生的 Java API，但是从 Observable 的角度使用它是很有诱惑力的。除了一个不同的 URI 之外，Camel 集成基本相同。

```
reactiveCamel
        .toObservable("kafka:localhost:9092?topic=demo&groupId=rx")
        .map(Message::getBody)
        .subscribe(e ->
                log.info("Message: {}", e));
```

我们可以消费来自几乎所有平台的抽象消息，它们使用相同的 ObservableAPI，这一点是非常强大的。Camel 在一致的接口背后提供了必要的物理连接，而 RxJava 通过大量的操作符进一步增强了该 API。在应用程序中，Camel 和 Retrofit（参见 8.1.2 节）是开始使用 Reactive Extensions 的良好起点。在拥有了稳定的 Observables 源之后，将反应式行为进一步扩展至技术栈就会容易得多。

8.5　Java 8的Stream和CompletableFuture

有时候，人们对于该选择哪种抽象进行并发编程可能会觉得困惑，在 Java 8 之后更是如此。多种互相竞争的 API 都能以非常整洁的方式表述异步计算。本节会对比这些 API，帮助你选择适合工作的恰当工具。可用的抽象包括以下部分。

- CompletableFuture

  CompletableFuture 由 Java 8 引入，这是对 java.util.concurrent 包中大家熟知的 Future 的强大扩展。CompletableFuture 允许 Future 完成或失败时注册异步回调，而不必阻塞等待结果。但是，它真正的强大之处在于组合和转换的能力，这类似于 Observable.map() 和 flatMap() 提供的功能。尽管 CompletableFuture 是在标准 JDK 中引入的，但是在 Java 标准库中并没有单个类依赖或使用它。虽然它十分好用，但是还没有很好地集成到 Java 生态系统中。要了解它与 RxJava 的可移植性问题，请参见 5.4.1 节。

- 并行（parallel）流

  与 CompletableFuture 类似，java.util.stream 中的流也是 JDK 8 引入的。流是一种能够在真正执行之前声明操作序列的方式，比如映射、过滤等。流上所有的操作都是延迟执行的，直到使用了像 collect() 或 reduce() 这样的终端操作才会真正执行。同时，JDK 能够自动在可用的核心上并行执行某些操作，这听起来非常有吸引力。并行流承诺能够在多个核心上透明地对较大的数据集进行映射、过滤，甚至排序。流通常由集合生成，但也可以在运行时动态创建元素数量无限的流。

- ❑ rx.Observable

  Observable 代表了一个事件流，而事件的出现时间是无法预测的。它可以代表零个、一个、固定数量或无限数量的事件，这些事件可以立即可用，也可以随时间推移而出现。完成事件或错误事件都可以终止 Observable。现在，你应该非常熟悉 Observable 了。

- ❑ rx.Single

  RxJava 库成熟之后，如果有一个特殊的类型能够代表有且仅有一个结果，那会有很大的助益。Single 类型的流要么有且仅有一个值并正常完成，要么出现错误。在某种程度上，Single 与 CompletableFuture 非常类似，但 Single 是延迟执行的，这意味着在订阅之前，它不会开始计算。5.5 节对 Single 进行了描述。

- ❑ rx.Completable

  有时候调用特定的计算纯粹是为了它的副作用，而不期望得到任何的结果。这样的例子包括发送 Email 或者保存数据库记录，这种操作涉及 I/O（这可以从异步处理中收益），但是并不会返回有意义的结果。传统上，这种场景会使用 CompletableFuture<Void> 或 Observable<Void>。但是，具体的 Completable 类型能够更清晰地表明执行异步计算却没有结果返回的意图。在并发执行中，Completable 会得到完成或错误通知，与其他的 Rx 类型相似，它也是延迟执行的。

显然，还有其他表述异步计算的方式，如下所示。

- Reactor 项目提供的 Flux 和 Mono。这两个类型分别类似于 Observable 和 Single。
- Guava 中的 ListenableFuture。

但是，为了使可选方案列表保持简短，我们可以将其限定在 JDK 和 RxJava 中。在开始之前，需要说明，如果应用程序已经非常一致地使用了 CompletableFuture，那么我们应该坚持使用它。CompletableFuture 提供的 API，有一些比较笨重，但是整体而言，这个类提供了对反应式编程的良好支持。另外，我们期望越来越多的框架能够利用这一点并提供惯用支持。在第三方库中支持 RxJava 会更加困难，因为这需要额外的依赖，而 CompletableFuture 是 JDK 的一部分。

8.5.1 并行流的用途

暂时转移一下话题，讨论一下来自标准 JDK 的并行流。在 Java 8 中，如果你需要转换一个较大的对象集合，可以声明以并行的方式进行转换。

```
List<Person> people = //...

List<String> sorted = people
    .parallelStream()
    .filter(p -> p.getAge() >= 18)
    .map(Person::getFirstName)
    .sorted(Comparator.comparing(String::toLowerCase))
    .collect(toList());
```

注意，上面的代码片段使用了 parallelStream()，而不是传统的 stream()。通过使用 parallelStream()，我们能够要求像 collect() 这样的终端操作以并行的方式执行，而不

是以序列化的方式。当然，这对结果不会有任何影响，但是速度会更快一些。在底层，parallelStream() 会将输入集合切分为多个块，并行处理每个块，然后按照分而治之的原则将结果组合在一起。

有些操作符很容易并行化（比如 map() 和 filter()），其他操作符可能会更困难一些（比如 sorted()）。因为在对每个块分别排序之后，必须再将它们组合在一起，在排序的场景下，这就意味着需要对两个有序的序列进行合并。如果没有进一步的假设条件，有些操作符很难或者无法并行化。例如，reduce() 只有在累加函数是可结合的时候，才能进行并行化。

相同的结果？

有些操作符在分别使用顺序化 stream() 和 parallelStream() 时可能产生不同的结果。例如，findFirst() 操作符会返回流中遇到的第一个元素。而 findAny() 操作符似乎完成的是相同的事情。但是，findFirst() 始终都会返回流中第一个元素，findAny() 则可以在并行流执行的时候返回流中任意一个值。

有可能发生这样的情况，在 findFirst() 或 findAny() 之前使用了 filter() 操作符。parallelStream() 的执行可以任意切分输入流，假设将其切分成了两部分，然后分别对这两部分执行过滤。如果对后半部分的过滤首先产生了匹配的值，findAny() 就会返回这个值，即使前半部分有匹配的值也会如此。findFirst() 能够保证返回全局第一个匹配的值，所以它必须等待这两部分的过滤结果。两个方法各有其优点，应该慎重使用。

在一台具备 4 个 CPU 的机器上应用 Amdahl 定律，理想情况下，执行的速度要快 4 倍。但是，并行流有自己的缺点。首先，如果流的规模比较小而且转换管道比较短，那么上下文切换的成本就会非常高，甚至可能出现并行流比顺序流更慢的情况。这种过于细粒度的并发问题在 RxJava 中也有可能出现，因此它支持通过 Scheduler（参见 4.9.1 节）进行声明式的并发。这种并行流的情况则有所不同。

有没有想过为什么这个框架叫作**并行**流而不是**并发**（concurrent）流？按照设计，并行流仅适用于 CPU 密集型的任务，它有一个硬编码的线程池（确切地说，是 ForkJoinPool），根据我们拥有的 CPU 数量进行分配。这个池可以通过 ForkJoinPool.commonPool() 全局静态访问。JVM 中的每个并行流以及一些 CompletableFuture 回调会共享该 ForkJoinPool。整个 JVM（所以，如果在应用程序服务器中部署多个 WAR 文件，这意味着会有多个应用程序）中的所有并行流共享同一个较小的池。这样是没有问题的，因为按照设计并行流用于并行的任务，它确实需要在这段时间内 CPU 达到 100%。因此，如果多个并行流并发执行，无论怎样它们肯定都会争用 CPU。

但是，假设有一个自私的应用程序，它在并行流中执行了 I/O 操作。

```
//不要这样做
people
        .parallelStream()
        .forEach(this::publishOverJms);
```

publishOverJms() 会为流中的每个人发送一条 JMS 消息。我们故意选择使用 JMS 进行发送。它看上去很快，但是为了保证投递成功，发送 JMS 很可能会涉及网络（通知消息代理）或磁盘（在本地持久化消息）。这个微小的 I/O 延迟足以占用宝贵的 ForkJoinPool.commonPool() 线程很长时间。即便这个程序没有使用 CPU，JVM 中的其他代码也无法执行并行流了。现在，设想一下，如果这个程序不是通过 JMS 发送消息，而是通过网络服务检索数据，或者运行代价高昂的数据库查询。parallelStream() 只能用于完全 CPU 密集的任务，否则，JVM 的性能会受到严重的影响。

这并不是说并行流不好。但是，线程池的数量是固定的，所以它们的应用场景较为有限。当然，来自 JDK 的并行流并不是 Observable.flatMap() 或其他并发机制的替代方案。并行流在并行执行的时候效果最好。但是，对于不要求 CPU 100% 占用的并发任务（比如会有网络或磁盘阻塞的场景），最好使用其他机制。

了解了流的局限性，接下来对比一下 future 和 RxJava，看看它们最适合什么样的场景。

8.5.2 选择恰当的并发抽象

在 RxJava 中，与 CompletableFuture 最接近的就是 Single 了。我们也可以使用 Observable，不过需要注意它会发布任意数量的值。future 和 RxJava 类型有一个很大的区别，即后者是延迟执行的。得到一个 CompletableFuture 引用时，我们就可以确定后台计算已经开始了，而 Single 和 Observable 很可能在订阅之后才会开始工作。了解了这种语义上的差异，我们就可以很容易地交替使用 CompletableFuture、Observable（参见 5.4 节）和 Single（参见 5.5.3 节）了。

在某些罕见情况下，无法获取异步计算的结果，或者结果本身无关紧要，那么可以使用 CompletableFuture<Void> 或 Observable<Void>。前者比较简单，而后者可能暗示着一个空事件形成的潜在无穷流。rx.Single<Void> 是一种糟糕的用法，与 Future 中返回的 Void 类似。因此，RxJava 引入了 rx.Completable。如果我们的架构有很多操作，却不会形成有意义的结果（不过可能会有异常），那么可以考虑使用 Completable。举例来说，在命令查询分离模式架构（Command-Query Separation，CQS）中，命令是异步的，而且按照定义它们不会返回任何结果。

8.5.3 何时使用Observable

如果我们的应用程序要处理随时间而出现的事件流（比如用户登录、GUI 事件以及推送通知），那么 Observable 是不二之选。本书虽然没有提及，但是 Java 从 1.0 版本就开始提供 java.util.Observable 了，它允许注册 Observer 和获取通知。不过，它在如下方面有所欠缺。

- 组合的能力（没有操作符）。
- 泛型（Observer 有一个 update() 方法，该方法接收 Object，代表任意的通知载荷）。
- 性能（到处使用 synchronized 关键字，内部使用 java.util.Vector）。
- 关注点分离（在某种意义上讲，它把 Observable 和 PublishSubject 组合在了一起）。
- 对并发的支持（所有的 Observer 都是顺序通知的）。

- 不可变性。

在标准 Java 中，JDK 中的 `Observable` 是声明式建模的事件的最好方案，其次是 GUI 包中的 `addListener()` 方法。如果领域明确涉及事件或数据流，那么很少有方案能够打败 `rx.Observable<T>`。声明式的表述方式再加上大量的操作符，能够解决大部分的问题。对于 cold 类型的 `Observable`，我们可以通过回压控制吞吐量；对于 hot 类型的 `Observable`，可以使用很多流控制操作符，比如 `buffer()`。

8.6　内存耗费和泄漏

RxJava 是关于事件流的，这些事件在运行时的内存中处理。它提供了一致且丰富的 API，将事件源的细节进行了抽象。理想情况下，在内存中应该只保留数量有限的一组固定事件，即从发布事件的生产者到存储或转发事件的消费者之间的事件。在现实中，有些组件，尤其是被误用的时候，消耗的内存可能不受任何限制。显然，内存是有限的，我们最终可能会遇到 `OutOfMemoryError` 或无休止的垃圾收集周期。本节将会介绍几个在 RxJava 中内存消耗不受控制和内存泄漏的样例，以及如何避免这种情况。Android 的相关章节介绍了一种特殊类型的内存泄漏，与忘记取消订阅相关，参见 8.1.1 节。

没有内存资源消耗限制的操作符

有些操作符可能会消耗任意数量的内存，消耗量完全取决于流的特性。接下来介绍几个这样的操作符，并尝试采取一些措施以避免内存泄漏。

1. distinct()缓存见过的所有事件

例如，按照定义，`distinct()` 必须存储自订阅以来见过的所有 key。`distinct()` 默认的重载版本会借助一个内部的缓存集，对比目前为止见过的所有事件。如果相同的事件（根据 `equals()` 方法进行判断）没有出现，这个事件会被发布，并且添加到缓存中以备未来使用。这个缓存永远不会被驱逐，[4] 以保证相同的事件不会再次出现。很容易想到，如果事件非常大或者事件出现得非常频繁，那么这个内部缓存会持续增长，从而导致内存泄漏。

为了方便阐述，使用如下事件模拟大块数据。

```
class Picture {
    private final byte[] blob = new byte[128 * 1024];
    private final long tag;

    Picture(long tag) { this.tag = tag; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Picture)) return false;
        Picture picture = (Picture) o;
```

注 4：它实际上是 RxJava1.1.6 中的 `HashSet`。

```
            return tag == picture.tag;
        }

        @Override
        public int hashCode() {
            return (int) (tag ^ (tag >>> 32));
        }

        @Override
        public String toString() {
            return Long.toString(tag);
        }
    }
```

下面的程序会在一个内存很有限的环境中运行（-mx32M：32 MB 的堆），并且尽可能快地发布非常大的事件。

```
    Observable
            .range(0, Integer.MAX_VALUE)
            .map(Picture::new)
            .distinct()
            .sample(1, TimeUnit.SECONDS)
            .subscribe(System.out::println);
```

运行之后，很快就会出现 OutOfMemoryError，因为 distinct() 内部的缓存无法盛放更多 Picture 实例。在崩溃之前，因为垃圾收集器努力释放空间，CPU 的使用率也非常高。但是，即便不使用整个 Picture 作为区分事件的 key，只使用 Picture.tag 进行区分，程序依然会崩溃，只不过出现的时间会晚很多。

```
    distinct(Picture::getTag)
```

这种类型的泄漏甚至更危险。在我们没有注意到的时候，这种问题变得越来越严重，直到在意料之外的时刻爆发，通常是在高负载的情况下。为了证明 distinct() 是内存泄漏的根源，可以运行一个类似的程序。该程序不使用 distinct()，而是统计在没有任何的缓冲的情况下每秒发布事件的数量。你的数量级可能会有所差异，但是下面的程序每秒能处理成千上万条大型消息，并不会对垃圾收集和内存带来太大压力。

```
    Observable
            .range(0, Integer.MAX_VALUE)
            .map(Picture::new)
            .window(1, TimeUnit.SECONDS)
            .flatMap(Observable::count)
            .subscribe(System.out::println);
```

那么，该如何避免 distinct() 相关的内存泄漏呢？

- 完全避免使用 distinct()。这种方法最简单，如果使用不当，这个操作符本身是很危险的。
- 明智地选择 key。理想情况下，key 应该占据有限且很小的空间。enum 和 byte 都是可以的，但是 long 或 String 可能不行。如果无法确定给定的类型是否只具有有限的值（像 enum 这样），那么可能会有内存泄漏的风险。

- 考虑使用 distinctUntilChanged() 作为替代方案，它只会跟踪最后一个见到的事件，而不是所有事件。
- 从一开始，我们真的需要唯一性吗，这个需求能否放松？或者能否以某种方式断定，重复的值只会在 10 秒之内出现？如果是这样，可以考虑在一个很小的窗口中运行 distinct()。

```
Observable
        .range(0, Integer.MAX_VALUE)
        .map(Picture::new)
        .window(10, TimeUnit.SECONDS)
        .flatMap(Observable::distinct)
```

样例每 10 秒会开启一个新的窗口（参见 6.1.2 节）并确保在这个窗口中没有重复的值。window() 会发布一个 Observable，这个 Observable 包含了每个时间窗口内的所有事件。在这个窗口中，唯一的值（通过 distinct() 判断）会立即发布。10 秒的窗口结束时，将开启一个新的窗口，但更重要的是，旧窗口关联的缓存会被垃圾收集。当然，在这 10 秒之内，依然有可能遇到数量较多的事件，从而导致 OutOfMemoryError。所以，我们最好采用固定长度的窗口（比如 window(1000)），而不是固定时间的窗口。另外，如果在一个窗口的结尾和另一个窗口的开端出现了相同的事件，那么样例将无法辨别重复的事件。这是我们必须要注意的权衡。

2. 通过toList()和buffer()缓冲事件

显然，toList() 会耗费无限的内存。另外，对无穷流使用 toList() 没有任何意义。toList() 只会在上游源完成的时候发布一个事件；如果上游不完成，toList() 不会发布任何内容。但是它会在内存中聚合所有事件。对很长的流使用 toList() 也是有问题的。我们应该以某种方式在运行时消费事件，或者使用像 take() 这样的操作符限制上游事件的数量。

如果你想要同时查看有限 Observable 中的所有事件，那么 toList() 是很有用的。这种情况很少见，我们可以使用谓词（如 allMatch() 和 anyMatch()）、计数（count()）或者将它们约减到一个聚合值（reduce()）中，这样能够避免同时在内存中保留所有的事件。如下的用例将 Observable<Observable<T>> 转换成 Observable<List<T>>，内部 Observable 的长度是固定的。

```
.window(100)
.flatMap(Observable::toList)
```

这等价于以下代码。

```
.buffer(100)
```

在使用 buffer() 之前，仔细思考一下，我们是否真的需要用某个时间段内所有事件组成的 List<T>。也许，使用一个 Observable<T> 就足够了。举例来说，如果我们想要知道每秒形成的 Observable<Incident> 中，高优先级的事件数量是否超过 5 件，那么需要生成一个 Observable<Boolean>，每秒都发布 true 或 false。如果给定的 1 秒内，发生了大量的高优先级的事件，那么会发布 true；否则，发布 false。借助 buffer()，这个功能实现起来非常简单。

```
Observable<Incident> incidents = //...

Observable<Boolean> danger = incidents
        .buffer(1, TimeUnit.SECONDS)
        .map((List<Incident> oneSecond) -> oneSecond
                .stream()
                .filter(Incident::isHIghPriority)
                .count() > 5);
```

但是，window() 并不需要将事件缓冲到一个中间 List 中，而是直接在运行时向下转发。window() 也能非常便利地完成相同的任务，而且保证了固定的内存消耗。

```
Observable<Boolean> danger = incidents
        .window(1, TimeUnit.SECONDS)
        .flatMap((Observable<Incident> oneSecond) ->
                oneSecond
                        .filter(Incident::isHIghPriority)
                        .count()
                        .map(c -> (c > 5))
        );
```

相对于 JDK 中的 Stream，Observable 有更加丰富的 API，所以你可能会发现，将 Java 的 Collection 转换成 Observable 只是为了能够使用更好的操作符。例如，Stream 并不支持滑动窗口和压缩。

所以，如果条件允许，我们应该优先使用 window()，而不是 buffer()，尤其是 buffer() 内部累积的 List 的长度无法预测和管理的时候。

3. 通过cache()和ReplaySubject进行缓存

cache() 是另外一个明显会消耗内存的操作符。比 distinct() 更糟糕的是，cache() 会保留从上游接收到的每个事件的引用。只有 Observable 的长度固定且元素数量较少的时候，使用 cache() 才有意义。例如，如果使用 Observable 来建模某个组件的异步响应，那么使用 cache() 是安全可行的；否则，Observer 会再次触发请求，可能会造成难以预料的副作用。相反，如果缓冲很长或无穷 Observable，尤其是 hot 类型的 Observable，使用 cache() 其实没有太大的意义。如果是 hot 类型的 Observable，我们其实很可能根本就不关心历史事件。

相同的情况适用于 ReplaySubject（参见 2.6 节）。这类 Subject 中的任何内容都必须进行存储，这样，后续的 Observer 能够看到所有的通知，而不仅仅是未来的通知。对 cache() 和 ReplaySubject 的建议几乎是相同的。如果你在使用它们，那么就要确保缓存源是有限的，并且元素的长度要相对较短。另外，不要长期保持对缓存 Observable 的引用；否则，一段时间之后可能会触发垃圾收集。

4. 回压能够保持内存使用处于较低的水平

3.2.3 节介绍了如何将两个事件生成节奏不同的源压缩在一起。如果想要压缩两个源，这两个源中有一个要比另一个稍慢一些，那么 zip()/zipWith() 操作符必须要临时缓冲较快的流，同时等待较慢的流所对应的事件。

```
Observable<Picture> fast = Observable
        .interval(10, MICROSECONDS)
        .map(Picture::new);
```

```
Observable<Picture> slow = Observable
        .interval(11, MICROSECONDS)
        .map(Picture::new);

Observable
        .zip(fast, slow, (f, s) -> f + " : " + s)
```

你可能认为上述代码会因为 OutOfMemoryError 而崩溃，认为 zip()[5] 要不断增加来自 fast 的事件的缓冲并等待 slow 流。但事实并非如此。实际上，样例几乎立即就会遇到恐怖的 MissingBackpressureException。zip()（和 zipWith()）操作符并不会不顾上游的吞吐量盲目地接收事件。相反，这些操作符会使用回压（参见 6.2 节），并且只请求尽可能少的数据。因此，如果上游 Observable 是 cold 类型的，并且实现得比较好，zip() 会减缓较快的 Observable。这是通过请求少量的数据实现的，而该 Observable 在技术上本来能够生成更多的数据。

而 interval() 的运行机制有所不同。interval() 操作符是 cold 类型的，只有有人订阅的时候才会开始计数，而且每个 Observer 都会得到独立的流。但是在订阅 interval() 之后，我们无法减缓它的速度，按照定义，它必须按照固定的频率发布事件。因此，它必须忽略回压请求，这可能会导致 MissingBackpressureException。我们能做的就是丢弃多余的事件（参见 6.2.3 节）。

```
Observable
        .zip(
                fast.onBackpressureDrop(),
                slow.onBackpressureDrop(),
                (f, s) -> f + " : " + s)
```

MissingBackpressureException 又比 OutOfMemoryError 好在哪里呢？缺失回压会立即失败，而内存不足则可能会缓慢累积，消耗本该分配到其他地方的宝贵内存。但是，缺失回压功能的异常也可能在最意想不到的时刻发生，比如在垃圾收集的时候。7.3 节讨论了如何对回压行为进行单元测试。

8.7　小结

如果我们的代码中已经有了一些 Observable 源，开始使用 RxJava 就会容易得多。从头实现新的 Observable 非常容易出错，所以当各种库（如 Hystrix,、Retrofit、数据库驱动）提供了对 RxJava 的原生支持后，使用 RxJava 就简单多了。在 4.1 节中，我们非常缓慢地将一个已有的应用程序从命令式、面向集合的风格重构为面向流、声明式的方式。但是在引入支持异步 Observable 源的库之后，重构就简单多了。应用程序中的流越多，反应式 API 就能向上传播地越广。从数据获取层（数据库、Web 服务等）开始，然后扩展至服务层和 Web 层，突然之间，我们会发现整个技术栈都是反应式的。从某种程度上来说，RxJava 的使用在项目中达到一个临界点之后，就不再需要 toBlocking() 了，因为从头到尾，所有的事情都变成了流。

注 5：在 RxJava 中引入回压前，zip() 就是这样工作的。我们很容易遇到异步的流，这会导致缓慢的内存泄漏。如果想了解 zip() 是如何重新实现的，那么需要在 0.20-RC2 版本中首先添加回压。

第 9 章

未来的方向

本 · 克里斯滕森（Ben Christensen）

在锁定 1.0 版本的 API 之前，RxJava 在 0.*x* 版本上阶段花费了很多时间，所以这是一个非常成熟和稳定的版本。同时，由于我们决定支持 API 上的 Experimental 和 Beta 标记，在将 API 提升至 Final 之前，一些正在进行中的实验性功能还会继续开展。但是，0.*x*/1.*x* 阶段中的一些决策依然存在具有破坏性的缺陷，所以 2.0 版本正在开发中。

基本上，它与 1.*x* 非常相似，所以不管在思想上还是用法上都不会有太大的变更。即便 2.0 版本已经发布，本书介绍的大部分内容依然有效。那么为什么要开发 2.0 版本呢？

9.1 反应式流

第一个原因是原生支持反应式流 API。尽管 RxJava 团队参与了反应式流的制订，但是 RxJava v1 的 API 已经被锁定，因此我们无法进行修改以适应反应式流中的接口。尽管 RxJava v1 在语义上的行为与反应式流非常相似，但是它需要一个适配器。2.0 版本会直接实现反应式流的类型并完全符合规范，以便更好地支持 Java 社区的互操作性。

9.2 `Observable`和`Flowable`

另外一个原因是将 `Observable` 类型分成了两个类型：`Observable` 和 `Flowable`。让一切都支持回压是错误的，并不是所有的用例都需要该特性。它的性能消耗很小，说它是一个错误的主要原因在于：它大幅增加了使用 `Observable` 和创建自定义操作符的难度。

单纯的推送场景应该按照 Erik Meijer 最初设计的那样使用 `Observable`，即不具备反应式流的 `request(n)` 语义。这样的用例非常普遍。基本上，所有的用户界面（user interface，UI）

用例（比如在 Android 中）都是纯推送的。在这种场景下，使用 `request(n)` 会令人产生困惑，增加不必要的复杂性。这时，`onBackpressureDrop` 风格的操作符可能会非常有用，但是应该有选择性地加入进来。

因此，在 v2 版本中，如果是单纯的推送，不支持 `request(n)` 就应该返回 `Observable`，它不会实现反应式流的类型或规范。同时，v2 版本新增加了一个类型 `Flowable`，它是“具备回压的 `Observable`”，实现了反应式流的 `Publisher` 类型和规范。将其命名为“Flowable”是受 Java 9 的 `java.util.concurrent.Flow` 的启发，后者采用了反应式流接口。

有了 `Observable` 和 `Flowable` 之后，就能够更好地在公开 API 交流该数据源的行为是什么。如果是 `Observable`，它支持推送，消费者必须已经准备就绪；如果是 `Flowable`，它执行拉取—推送模式，只会发送消费者请求数量的条目。类似于 RxJava v1 中做的那样，它们两者之间也可以转换，不过此时的转换会更加明确，比如 `observable.toFlowable(Strategy.DROP)`。这样就会将 `Observable` 转换为具有对应回压策略的 `Flowable`，如果数据的推送速度比消费者的处理速度更快，会应用该策略。

9.3 性能

开发 v2 版本的最后一个主要原因是提升整体的性能（减少资源消耗），从而规避 v1 版本的架构限制。这在一定程度上是通过减少在构建操作符链、订阅和运行它们时分配的数量实现的。默认情况下，`Subscriber` 不会再包装到一个 `SafeSubscriber` 中（为此提供了 `Flowable.safeSubscribe()`），终端事件上也不再需要通过 `cancel`（在 v2 版本的语义中，对应的是 `unsubscribe`）取消这个链。

性能改进的第二个来源是一种内部优化方法，称为**操作符融合**（operator-fusion，它扩展了反应式流协议）。在很多同步流的环境中，它能够大幅减少回压和队列管理的消耗（在某些异步流中也能达到该效果）。在一些基准测试中，开启了回压功能的流在吞吐量上仅比 Java 8 的 Stream（是同步拉取的）实现慢 20%~30%，而在 v1 版本中要慢 100%~200%。

9.4 迁移

由于 RxJava 在应用程序中影响较大，难以接受破坏性的变更。因此，v2 版本使用了不同的包名和 Maven artifact ID（表 9-1），这样 v1 和 v2 版本在同一个项目中就能够共存了。

表9-1

v1的包	v2的包	v1的Maven	v2的Maven
`rx.*`	`io.reactivex.*`	`io.reactivex:rxjava`	`io.reactivex.rxjava2:rxjava`

从 RxJava 的 v1 版本切换至 v2 版本主要涉及如下工作。

(1) 将包名从 `rx.` 替换成 `io.reactivex`。
(2) 如果需要回压功能，将 `Observable` 替换为 `Flowable`。

RxJava v2 位于 GitHub 的 2.*x* 分支，DESIGN.md 文档介绍了社区设计 v2 版本时在技术决策方面做出的努力。关于 v1 和 v2 版本差异的更多信息可以在 GitHub 上找到。

附录A

HTTP服务器的更多样例

附录 A 对 5.1 节的内容进行了扩展，提供了更多的 HTTP 服务器样例。这些样例对于理解第 5 章的内容并不是必需的，但是我们可能会发现它们的有趣之处。另外，这些样例中的一部分也包含在了基准测试中。

A.1 C语言中的fork()程序

我们尝试使用 C 语言实现一个并发 HTTP 服务器。如果你熟悉 C 语言，会发现如下的程序非常简单。如果不熟悉也不用担心，你不必了解全部的细节，掌握整体的理念即可。调用 `fork()` 会生成当前进程的一个副本，以至于在操作系统中会突然出现两个进程：原始的进程（父进程）和子进程。第二个进程具有完全相同的变量和状态，唯一的差异是 `fork()` 的结果值。

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  signal(SIGCHLD, SIG_IGN);
  struct sockaddr_in serv_addr;
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(8080);
  int server_socket = socket(AF_INET, SOCK_STREAM, 0);
  if(server_socket < 0) {
```

```
    perror("socket");
    exit(1);
  }
  if(bind(server_socket,
        (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0) {
    perror("bind");
    exit(1);
  }
  listen(server_socket, 100);
  struct sockaddr_in cli_addr;
  socklen_t clilen = sizeof(cli_addr);
  while (1) {
    int client_socket = accept(
        server_socket, (struct sockaddr *) &cli_addr, &clilen);
    if(client_socket < 0) {
      perror("accept");
      exit(1);
    }
    int pid = fork();
    if (pid == 0) {
      close(server_socket);
      char buffer[1024];
      while(1) {
        if(read(client_socket,buffer,255) < 0) {
          perror("read");
          exit(1);
        }
        if(write(client_socket,
          "HTTP/1.1 200 OK\r\nContent-length: 2\r\n\r\nOK",
          40) < 0) {
          perror("write");
          exit(1);
        }
      }
    } else {
      if(pid < 0) {
        perror("fork");
        exit(1);
      }
    }
    close(client_socket);
  }
  return 0;
}
```

真正重要的是对 fork() 的调用。在父进程（原始进程）中，它返回了子进程的 PID（进程 ID）；在子进程（副本进程）中，它返回的是 0。在某些情况下，fork() 只执行一次（在父进程中），但是会返回两次。如果发现自己是子进程（fork() == 0），那么就需要处理客户端连接。server_socket 由父进程管理，可以在子进程中关闭。同时（并发地），父进程关闭 client_socket（子进程依然使其处于打开状态），并且能够通过 accept() 接收其他客户端连接。当然，父线程可以同时派生（fork）多个子线程，以实现更高的并发性。

A.2 每个连接对应一个线程

既然一个线程难以很好地扩展服务器（参见 5.1.1 节），那么我们就采取一些线程技术对其进行重写。在进入实现细节之前，我们先重写一下 SingleThread 类，避免在后续样例中重复出现。

```
abstract class HttpServer  {

    void run(int port) throws IOException {
        final ServerSocket serverSocket = new ServerSocket(port, 100);
        while (!Thread.currentThread().isInterrupted()) {
            final Socket client = serverSocket.accept();
            handle(new ClientConnection(client));
        }
    }

    abstract void handle(ClientConnection clientConnection);
}
```

ClientConnection 类如下所示。

```
import org.apache.commons.io.IOUtils;

class ClientConnection implements Runnable {

    public static final byte[] RESPONSE = (
            "HTTP/1.1 200 OK\r\n" +
            "Content-length: 2\r\n" +
            "\r\n" +
            "OK").getBytes();

    public static final byte[] SERVICE_UNAVAILABLE = (
            "HTTP/1.1 503 Service unavailable\r\n").getBytes();

    private final Socket client;

    ClientConnection(Socket client) {
        this.client = client;
    }

    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                readFullRequest();
                client.getOutputStream().write(RESPONSE);
            }
        } catch (Exception e) {
            e.printStackTrace();
            IOUtils.closeQuietly(client);
        }
    }

    private void readFullRequest() throws IOException {
        BufferedReader reader = new BufferedReader(
```

```
                new InputStreamReader(client.getInputStream()));
        String line = reader.readLine();
        while (line != null && !line.isEmpty()) {
            line = reader.readLine();
        }
    }

    public void serviceUnavailable() {
        try {
            client.getOutputStream().write(SERVICE_UNAVAILABLE);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

}
```

这只是一个简单的重构：我们将一些通用的样板代码（比如在循环中监听客户端的连接）转移到基类。同时，将处理客户端连接的功能放到一个单独的 `ClientConnection` 类中，随后使用一个额外的 `serviceUnavailable()`。`HttpServer` 实现的唯一责任就是以某种方法调用 `ClientConnection` 的 `run()`，例如在重构后的 `SingleThread` 中直接进行调用。

```
public class SingleThread extends HttpServer {

    public static void main(String[] args) throws Exception {
        new SingleThread().run(8080);
    }

    @Override
    void handle(ClientConnection clientConnection) {
        clientConnection.run();
    }
}
```

有了基础**框架**（framework），我们就可以快速构建更具扩展性的实现，这种实现方式会为每个 `ClientConnection` 生成一个新的 `Thread`。

```
public class ThreadPerConnection extends HttpServer {

    public static void main(String[] args) throws IOException {
        new ThreadPerConnection().run(8080);
    }

    @Override
    void handle(ClientConnection clientConnection) {
        new Thread(clientConnection).start();
    }
}
```

利用 `ClientConnection` 也是 `Runnable` 这个事实，样例为每个新连接只启动了一个 `Thread`。现在，服务器被缓慢的客户端阻塞的问题得到了缓解：对连接的处理在后台发生。这样，当针对客户端 `Socket` 读取和写入数据时，主线程依然能够接收新的连接。当然，如果同时有两个客户端连接，主线程会启动两个后台线程并继续后面的操作。

毫无限制地创建新线程也有一些缺点。在 64 位的 JVM 1.8 上，每个线程默认消耗 RAM 上 1024 KB 的空间（参见 `-Xss` 标记）。如果有上千个并发连接，即便它们是空闲的，也意味着会有 1000 个线程和 1 GB 的栈空间。不要混淆，栈空间独立于堆空间，所以应用程序消耗的内存远超过 1 GB。

A.3 连接的线程池

这次创建一个由空闲线程组成的池，让它等待传入的连接。包装了客户端 `Socket` 的新 `ClientConnection` 出现时，我们会使用池中第一个空闲线程。相对于按需创建线程，线程池的方式有以下优点。

- `Thread` 已经初始化和启动完成，因此不必等待或预热，减少了客户端的延迟。
- 系统中的线程总数有严格限制，因此在高峰负载时可以安全地拒绝连接，而不会引起系统崩溃。
- 线程池有一个可配置的队列，可以临时存放短期高峰时的负载。
- 如果池和队列都已经饱和，那么还可以使用一个可配置的拒绝策略（报错，转而在客户端线程中运行等）。

如果想完全控制正在创建的线程，那么相对于每次都创建新线程，线程池是一种更好的方式。更重要的是，我们可以严格控制客户端线程的总数，并且管理峰值状态。

```
class ThreadPool extends HttpServer {

    private final ThreadPoolExecutor executor;

    public static void main(String[] args) throws IOException {
        new ThreadPool().run(8080);
    }

    public ThreadPool() {
        BlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(1000);
        executor = new ThreadPoolExecutor(100, 100, 0L,
                                MILLISECONDS, workQueue,
                (r, ex) -> {
                    ((ClientConnection) r).serviceUnavailable();
                });
    }

    @Override
    void handle(ClientConnection clientConnection) {
        executor.execute(clientConnection);
    }

}
```

需要处理 `ClientConnection` 的时候，我们可以将这个任务移交给专门的 `ThreadPoolExecutor`，它在内部管理 100 个线程。在这个池的前面有一个有界的队列（1000 个任务），在出现大量请求时，`RejectedExecutionHandler` 就会发挥作用。服务器会简单地调用 `serviceUnavailable()`，并立即为客户端返回 503（快速失败行为，参见 8.2 节），而不是让客户端无休止地等待。

借助 Servlet 3.0 规范，我们能够基于异步 Servlet 编写可扩展的应用程序。它的理念是将请求的处理和容器线程进行解耦。当请求需要发送响应时，它可以在任何时间点的任何线程中发送。而接收请求的原始容器线程可能已经消失，或在处理其他请求。这是一个革命性的理念，但是应用程序的其他部分也必须按照这种方式进行构建。否则，应用程序可能会更具响应性（容器线程池很少会饱和）。但是如果必须有另外的用户线程处理该请求，那么只是将线程暴增的问题转移到了其他地方而已。线程数量达到数百甚至数千的时候，应用程序的行为就会出现异常，例如，由于频繁的垃圾收集循环和上下文切换，它的响应变得越来越慢。

附录B

Observable操作符的决策树

本附录旨在帮助你从 RxJava 中找到合适的操作符。我们有上百个可选项，找到最适合需求的内置操作符会变得越来越复杂。附录的内容完全复制自 RxJava 官方文档，这个 `Observable` 操作符决策树文档基于 Apache 2.0 许可证。但是，接下来的反向引用指的是本书中的章节，而不是在线文档。大多数给定操作符都有一个完整的章节，有些操作符只进行简单的介绍或提供一个样例。

- 我想要创建一个新的 `Observable`……
  - 想要发布特定的条目，使用 `just()`，参见 2.4 节……
    - 想要在订阅的时候，从一个函数返回内容，使用 `start()`，参见 `rxjava-async` 模块。
    - 想要在订阅的时候，通过 `Action`、`Callable` 或 `Runnable` 等返回内容，使用 `from()`、`fromCallable()` 或 `fromRunnable()`，参见 2.4 节和 2.4.2 节。
    - 在特定的延迟后生成，使用 `timer()`，参见 2.4.3 节。
  - 想要从特定的 `Array`、`Iterable` 等发布内容，使用 `from()`，参见 2.4 节。
  - 想要从 Future 获取内容，使用 `from()`，参见 2.4 节和 5.4 节。
  - 想要从 Future 获取序列，使用 `from()`，参见 2.4 节。
  - 想要重复性地发布条目序列，使用 `repeat()`，参见 3.5.1 节。
  - 想要从头开始使用自定义的逻辑发布内容，使用 `create()`，参见 2.4.1 节。
  - 想要为订阅的每个观察者发布内容，使用 `defer()`，参见 4.3 节。
  - 想要发布整数序列，使用 `range()`，参见 2.4 节……
    - 想要按照特定的时间间隔生成，使用 `interval()`，参见 2.4.3 节。
      - 在特定延迟之后生成，使用 `timer()`，参见 2.4.3 节。
  - 想要不发布任何条目就结束，使用 `empty()`，参见 2.4 节。
  - 想要 `Observable` 什么事情都不做，使用 `never()`，参见 2.4 节。
- 我想要通过联合其他 `Observable` 的方式创建 `Observable`……

- 想要将所有 Observable 的条目按照接收到的顺序发布，使用 merge()，参见 3.2.1 节。
- 想要将所有 Observable 的条目都发布出去，但是每次只发布一个 Observable 的条目，使用 concat()，参见 3.4.1 节。
- 想要将两个或更多 Observable 中的条目顺序地组合在一起生成新条目，并且发布该条目……
  - 无论何时**每个** Observable 都发布了一个新条目，使用 zip()，参见 3.2.2 节。
  - 无论何时**任意**一个 Observable 发布一个新条目，使用 combineLatest()，参见 3.2.3 节。
  - 通过 Pattern 和 Plan 的中介 And/Then/when()，参见 rxjava-joins 模块。
- 只从最近发布的一个 Observable 中发布条目，使用 switch()，参见 3.4.1 节。

- 我想要将 Observable 中的条目转换之后再发布……
  - 以调用函数的形式，每次发布一个条目，使用 map()，参见 3.1 节。
  - 通过对应 Observable 发布所有条目，使用 flatMap()，参见 3.1.2 节。
    - 每次生成一个 Observable，按照发布的顺序进行排列，使用 concatMap()，参见 3.1.5 节。
  - 基于它们前面的所有条目，使用 scan()，参见 3.3.1 节。
  - 通过为条目附加一个时间戳，使用 timestamp()，参见 3.2.3 节。
  - 转换为一个指示器，表明该条目发布之前已经消耗的时间，使用 timeInterval()，参见 7.1.3 节。
- 我想要将 Observable 发布的条目及时转发后，再重新发布，使用 delay()，参见 3.1.3 节。
- 我想要将 Observable 中的条目**和**通知转换为条目并重新发布……
  - 使用 materialize() 将其包装到 Notification 对象中，参见 7.3 节。
    - 使用 dematerialize() 再次进行拆解。
- 我想要忽略 Observable 发布的所有条目，只向下传播完成 / 错误通知，使用 ignoreElements()，参见 4.6 节。
- 我想要得到 Observable 的镜像，但是在真正的序列之前添加一些条目，使用 startWith()，参见 3.2.3 节……
  - 仅当 Observable 为空时，才添加这些条目，使用 defaultIfEmpty()。
- 我想要收集某个 Observable 的所有条目，并以缓冲的方式重新发布，使用 buffer()，参见 8.6 节。
  - 只想包含最后发布的一些条目，使用 takeLastBuffer()。
- 我想要将一个 Observable 分割为多个 Observable，使用 window()，参见 6.1.3 节。
  - 想要将相似的条目最终放到同一个 Observable，使用 groupBy()，参见 3.4.2 节。
- 我想要检索 Observable 发布的一些特定条目……
  - 想要完成之前的最后一个条目，使用 last()，参见 3.4 节。
  - 想要发布的唯一条目，使用 single()，参见 3.3.3 节。
  - 想要发布的第一个条目，使用 first()，参见 3.4 节。
- 我想要重新发布 Observable 中的特定条目……
  - 过滤不符合指定断言的条目，使用 filter()，参见 3.1 节。
  - 只要第一个条目，使用 first()，参见 3.4 节。
  - 只要前几个条目，使用 take()，参见 3.4 节。
  - 只要最后一个条目，使用 last()，参见 3.4 节。
  - 只要排序为 *n* 的条目，使用 elementAt()，参见 3.4 节。

- 只要前几个条目之后的给定条目……
  - 跳过前 *n* 个条目，使用 skip()，参见 3.4 节。
  - 直到其中的一个条目符合指定的断言，使用 skipWhile()，参见 7.1.3 节。
  - 获取初始的时间段之后的条目，使用 skip()。
  - 第二个 Observable 之后开始发布条目，使用 skipUntil()。
- 想要得到最后这些条目之外的其他条目……
  - 不想要最后的 *n* 个条目，使用 skipLast()，参见 3.4 节。
  - 直到其中的一个条目符合指定的断言，使用 takeWhile()，参见 3.4 节。
  - 不想要源完成之前那段时间内的条目，使用 skipLast()。
  - 不想要第二个 Observable 发布的条目之后的所有条目，使用 takeUntil()。
- 想要对 Observable 进行周期性采样，使用 sample()，参见 6.1.1 节。
- 只想要发布条目间隔时间不在一定时间范围内的事件,使用 debounce(),参见 6.1.4 节。
- 想要跳过已发布的重复条目，使用 distinct()，参见 3.3.4 节……
  - 想要跳过紧邻的重复条目，使用 distinctUntilChanged()，参见 3.3.4 节。
- 想要在 Observable 发布条目之后，延迟对它的订阅，使用 delaySubscription()。

附录C

RxJava 1.0至RxJava 2.0的迁移指南

在翻译本书时，RxJava 2.0 已经推出，且 RxJava 1.0 已终止维护。尽管 RxJava 1.0 与 RxJava 2.0 在理念上一脉相承，但 RxJava 2.0 完全是重新编写的，所以在一些具体用法上会有较大的差异。因此，经许可，译者翻译了 GitHub 上关于 RxJava 1.0 与 RxJava 2.0 差异的两篇文章，供读者阅读参考。这两篇文章的作者是 RxJava 2.0 的核心贡献者和负责人 Dávid Karnok。其中，第一篇文章概要阐述了两个版本的差异，第二篇文章主要阐述了在 RxJava 2.0 版本中如何使用回压功能。

C.1　RxJava 2.0版本的变化

RxJava 2.0 是在反应式流规范之上完全重写的。这个规范本身从 RxJava 1.*x* 发展而来，为反应式流和库提供了一个通用的基准。

因为反应式流具有不同的架构，所以需要对一些众所周知的 RxJava 类型做出变更。本文会尝试总结这些变更，并描述如何将 RxJava 1.*x* 的代码重写为 RxJava 2.*x* 的代码。

关于 2.*x* 编写操作符的更多技术细节，请参阅 GitHub 上的文章 *Writing Operators*。

C.1.1　Maven地址和基础包

为了让 RxJava 1.*x* 和 RxJava 2.*x* 共存，RxJava 2.*x* 的 Maven 坐标为 `io.reactivex.rxjava2:rxjava:2.x.y`，类则位于 `io.reactivex` 包下。

要想从 RxJava 1.*x* 切换至 RxJava 2.*x*，需要重新组织导入语句，不过要小心一些。

C.1.2　Javadoc

针对 2.*x* 的 Javadoc 页面现在托管在 http://reactivex.io/RxJava/2.x/javadoc/ 中。

C.1.3　关于null

RxJava 2.*x* 不能接受 null 值。因此，如下代码会立即抛出 NullPointerException，或者作为错误信号传递到下游中。

```
Observable.just(null);

Single.just(null);

Observable.fromCallable(() -> null)
    .subscribe(System.out::println, Throwable::printStackTrace);

Observable.just(1).map(v -> null)
    .subscribe(System.out::println, Throwable::printStackTrace);
```

这意味着 Observable<Void> 不会发布任何值，只能正常终止或者以异常的方式终止。API 设计者反而可以选择将 API 定义为 Observable<Object>，但是这样将无法对 Object 的内容提供保证（不过，应该关系不大）。例如，如果需要类似信号的源，那么可以定义一个共享的枚举并在 onNext 中使用该单实例的枚举值。

```
enum Irrelevant { INSTANCE; }

Observable<Object> source = Observable.create((ObservableEmitter<Object> emitter) -> {
    System.out.println("Side-effect 1");
    emitter.onNext(Irrelevant.INSTANCE);

    System.out.println("Side-effect 2");
    emitter.onNext(Irrelevant.INSTANCE);

    System.out.println("Side-effect 3");
    emitter.onNext(Irrelevant.INSTANCE);
});

source.subscribe(e -> { /* Ignored. */ }, Throwable::printStackTrace);
```

C.1.4　Observable和Flowable

在 RxJava 0.*x* 中引入回压有一个小遗憾，即没有创建单独的反应式基础类，而是直接改造了 Observable 本身。回压的主要问题在于很多 hot 类型的源（比如 UI 事件）不能合理地添加回压功能，因此会造成 MissingBackpressureException。

2.*x* 版本试图弥补这一点，同时提供了不支持回压的 io.reactivex.Observable 以及支持回压的新反应式基础类 io.reactivex.Flowable。

好消息是操作符的名称（大部分）都保持了一致；坏消息是组织导入语句时要十分小心，因为有可能会导入不支持回压的 io.reactivex.Observable。

使用哪种类型

在对数据流进行架构设计（作为 RxJava 的终端消费者），或者编写兼容 RxJava 2.*x* 版本库的代码时，如果需要决定使用和返回的类型，那么可以考量一些因素，避免出现

MissingBackpressureException 或 OutOfMemoryError 这样的问题。

- **何时使用 Observable**
  - 如果流的最大长度不超过 1000 个元素，也就是说，随着时间的推移，元素的数量很少，应用程序基本上不会出现 OOME 错误。
  - 如果处理的是 GUI 事件，比如鼠标移动或触摸事件：这些事件很少能合理地实现回压功能，而且它们的出现频率不会很高。借助 Observable，能够处理频率为 1000 Hz 或更低的元素，但是可以考虑使用采样 / 去除抖动的策略。
  - 如果你的流在本质上是同步的，但是平台不支持 Java Stream 或者不能使用某些特性。使用 Observable 的开销一般比 Flowable 低一些（也可以考虑使用 IxJava，它对 Iterable 流进行了优化，支持 Java 6+）。
- **何时使用 Flowable**
  - 要处理的元素数量超过 10 000 个，因此能够告诉源限制生成的元素数量。
  - 从磁盘读取（解析）文件本质上是阻塞的，并且是基于拉取模式的。它能够很好地支持回压，比如针对请求所要求的数量，控制读取多少行。
  - 通过 JDBC 读取数据库也是阻塞的，并且是基于拉取模式的，可以通过对每个下游请求调用 ResultSet.next() 来进行控制。
  - 网络（流式）IO 的场景下，如果网络或者所使用的协议能够支持请求一定的逻辑数量的元素。 无论是网络帮助还是使用的协议支持都请求一定的逻辑数量。
  - 很多阻塞式和基于拉取模式的数据源，最终可能会在未来获得非阻塞的反应式 API/驱动。

C.1.5 Single

2.*x* 中的 Single 反应式基础类型进行了重新设计，能够发布一个 onSuccess 或 onError 通知。它的架构来源于反应式流的设计。它的消费者类型（rx.Single.SingleSubscriber<T>）已经从接收 rx.Subscription 资源的类变更为 io.reactivex.SingleObserver<T> 接口，该接口只有 3 种方法。

```
interface SingleObserver<T> {
    void onSubscribe(Disposable d);
    void onSuccess(T value);
    void onError(Throwable error);
}
```

它遵循的协议为：onSubscribe (onSuccess | onError)?。

C.1.6 Completable

Completable 类型基本上和原来相同，1.*x* 版本的设计就遵循了反应式流的风格，因此没有用户级别的变更。

名称的变化非常类似，rx.Completable.CompletableSubscriber 变成了 io.reactivex.CompletableObserver，后者现在具有 onSubscribe(Disposable) 方法。

```
interface CompletableObserver<T> {
    void onSubscribe(Disposable d);
    void onComplete();
    void onError(Throwable error);
}
```

它遵循的协议依然为：onSubscribe (onComplete| onError)?。

C.1.7 Maybe

RxJava 2.0.0-RC2 引入了一个新的反应式基础类型 Maybe。从概念上讲，它是 Single 和 Completable 的组合体，能捕获的发布模式是 0 个条目、1 个条目或来自反应式源的一个错误。

Maybe 类的基础接口类型为 MaybeSource，其信号接收接口为 MaybeObserver<T>，它遵循的协议为 onSubscribe (onSuccess | onError | onComplete)?。因为可能最多只发布一个元素，所以 Maybe 类型没有回压的理念（它不可能像 Flowable 或 Observable 那样，因为未知的元素长度带来缓冲区的膨胀）。

这意味着调用 onSubscribe(Disposable) 之后，可能会紧跟着调用一个其他的 onXXX 方法。与 Flowable 不同，如果仅发布一个值，这里只会调用 onSuccess，而不会调用 onComplete。

使用这个新的反应式基础类型和使用其他类型几乎相同，因为它是 Flowable 操作符的一个子集，这些操作符适用于 0 个或 1 个条目组成的序列。

```
Maybe.just(1)
.map(v -> v + 1)
.filter(v -> v == 1)
.defaultIfEmpty(2)
.test()
.assertResult(2);
```

C.1.8 基础的反应式接口

与 Flowable 中的扩展反应式流的 Publisher<T> 风格类似，其他的反应式基础类也扩展了类似的基础接口（在 io.reactivex 包中）。

```
interface ObservableSource<T> {
    void subscribe(Observer<? super T> observer);
}

interface SingleSource<T> {
    void subscribe(SingleObserver<? super T> observer);
}

interface CompletableSource {
    void subscribe(CompletableObserver observer);
}

interface MaybeSource<T> {
    void subscribe(MaybeObserver<? super T> observer);
}
```

因此，很多需要反应式基础类型的操作符可以接收 Publisher 和 XSource 了。

```
Flowable<R> flatMap(Function<? super T, ? extends Publisher<? extends R>> mapper);

Observable<R> flatMap(Function<? super T, ? extends ObservableSource<? extends R>>
mapper);
```

通过将 Publisher 作为输入，可以组合其他兼容反应式流的库，而不需要再将它们进行包装并转换为 Flowable 了。

但是，如果一个操作符必须要提供反应式基础类型，那么用户会接收到完整的反应式类（给用户分发一个 XSource 其实没有什么用处，因为没有它的操作符）。

```
Flowable<Flowable<Integer>> windows = source.window(5);

source.compose((Flowable<T> flowable) ->
    flowable
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread()));
```

C.1.9 Subject和Processor

在反应式流规范中，类似 Subject 的行为（即同时是事件的消费者和提供者）已经通过 org.reactivestreams.Processor 接口实现了。随着 Observable/Flowable 的拆分，能够感知回压、符合反应式流规范的实现是基于 FlowableProcessor<T> 类的（它扩展了 Flowable 以提供丰富的实例操作符）。关于 Subject（以及它的扩展 FlowableProcessor）还有一个重要的变化：它们不再支持类似 T -> R 的转换（即输入 T 类型，输出 R 类型）。1.*x* 中从未使用过它，而原始的 Subject<T, R> 来自 .NET，.NET 有一个 Subject<T> 重载，因为 .NET 允许相同的类名和不同数量的类型参数。

在 2.*x* 中，io.reactivex.subjects.AsyncSubject、io.reactivex.subjects.BehaviorSubject、io.reactivex.subjects.PublishSubject、io.reactivex.subjects.ReplaySubject 和 io.reactivex.subjects.UnicastSubject 不支持回压（因为它们是 Observable 家族的一部分）。

io.reactivex.processors.AsyncProcessor、io.reactivex.processors.BehaviorProcessor、io.reactivex.processors.PublishProcessor、io.reactivex.processors.ReplayProcessor 和 io.reactivex.processors.UnicastProcessor 能够感知回压。BehaviorProcessor 和 PublishProcessor 不会协调下游订阅者请求（这一点要通过 Flowable.publish() 来实现），如果下游的处理节奏跟不上，会以 MissingBackpressureException 的形式通知它们。其他的 XProcessor 类型在回压方面能够遵循下游订阅者的请求量，但是当订阅某个源（可选的）时，它们会以无界的方式进行消费（请求 Long.MAX_VALUE）。

1. TestSubject

1.*x* 版本的 TestSubject 已经废弃了。它的功能可以通过 TestScheduler、PublishProcessor/PublishSubject 和 observeOn(testScheduler)/ 调度器参数实现。

```
TestScheduler scheduler = new TestScheduler();
PublishSubject<Integer> ps = PublishSubject.create();

TestObserver<Integer> ts = ps.delay(1000, TimeUnit.MILLISECONDS, scheduler)
.test();
```

```
ts.assertEmpty();

ps.onNext(1);

scheduler.advanceTimeBy(999, TimeUnit.MILLISECONDS);

ts.assertEmpty();

scheduler.advanceTimeBy(1, TimeUnit.MILLISECONDS);

ts.assertValue(1);
```

2. SerializedSubject

SerializedSubject 不再是一个公开类（public class），必须要使用 Subject.toSerialized() 和 FlowableProcessor.toSerialized() 作为替代。

C.1.10　其他的类

rx.observables.ConnectableObservable 拆分为 io.reactivex.observables.ConnectableObservable<T> 和 io.reactivex.flowables.ConnectableFlowable<T>。

GroupedObservable

rx.observables.GroupedObservable 拆分为 io.reactivex.observables.GroupedObservable<T> 和 io.reactivex.flowables.GroupedFlowable<T>。

在 1.*x* 中，你可以通过 GroupedObservable.from() 创建一个只在 1.*x* 内部使用的实例。在 2.*x* 中，所有的用例都可以直接扩展 GroupedObservable，所以工厂方法没有必要存在了，整个类现在变成了抽象的。

我们可以直接扩展这个类并添加自定义的 subscribeActual 行为，实现类似 1.*x* 的特性。

```
class MyGroup<K, V> extends GroupedObservable<K, V> {
    final K key;

    final Subject<V> subject;

    public MyGroup(K key) {
        this.key = key;
        this.subject = PublishSubject.create();
    }

    @Override
    public T getKey() {
        return key;
    }

    @Override
    protected void subscribeActual(Observer<? super T> observer) {
        subject.subscribe(observer);
    }
}
```

（这一方式对于 GroupedFlowable 也是有效的。）

C.1.11 函数式接口

因为 1.*x* 和 2.*x* 面向的都是 Java 6+，所以无法使用 Java 8 的函数式接口，比如 `java.util.function.Function`。于是，1.*x* 定义了自己的函数式接口，2.*x* 延续了这一传统。

很重要的一个变化就是所有的函数式接口都定义了 `throws Exception`。这对于消费者和映射器是非常便利的，否则，如果抛出异常，就需要 `try-catch` 语句来转换或抑制检查型异常。

```
Flowable.just("file.txt")
.map(name -> Files.readLines(name))
.subscribe(lines -> System.out.println(lines.size()), Throwable::printStackTrace);
```

如果文件不存在或者无法正确读取，那么终端用户会直接打印出 `IOException`。注意，在没有 `try-catch` 的情况下，`Files.readLines(name)` 也被调用了。

1. Action

鉴于这是一个减少组件数量的机会，2.*x* 没有定义 `Action3-Action9` 以及 `ActionN`（RxJava 本身没有用到它们）。

剩余接口的名称按照相应的 Java 8 函数式类型进行了命名。无参的 `Action0` 由 `io.reactivex.functions.Action` 替换以用于各种操作符，而 `Schedule` 方法则会使用 `java.lang.Runnable`。`Action1` 重命名为 `Consumer`，`Action2` 则称为 `BiConsumer`，`ActionN` 替换为 `Consumer<Object[]>` 类型声明。

2. Function

我们遵循 Java 8 的命名约定，定义了 `io.reactivex.functions.Function` 和 `io.reactivex.functions.BiFunction`，同时相应地将 `Func3-Func9` 重命名为 `Function3-Function9`。`FuncN` 替换为 `Function<Object[], R>` 类型声明。

除此之外，需要断言的操作符现在不再使用 `Function<Object[], R>` 了，而是有了一个单独的、返回原始类型的 `Predicate<T>`（因为没有自动拆箱，能够实现更好的内联）。

C.1.12 Subscriber

反应式流规范有自己的 `Subscriber` 接口。这个接口是轻量级的，它将请求管理和取消放到了一个接口中，也就是 `org.reactivestreams.Subscription`，而不是分别使用 `rx.Producer` 和 `rx.Subscription`。这样，相对于 1.*x* 中重量级的 `rx.Subscriber`，2.*x* 在创建流的消费者时可以使用更少的内部状态。

```
Flowable.range(1, 10).subscribe(new Subscriber<Integer>() {
    @Override
    public void onSubscribe(Subscription s) {
        s.request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(Integer t) {
        System.out.println(t);
```

```
        }

        @Override
        public void onError(Throwable t) {
            t.printStackTrace();
        }

        @Override
        public void onComplete() {
            System.out.println("Done");
        }
    });
```

因为名称的冲突，单纯地将包名从 rx 替换为 org.reactivestreams 是不够的。除此之外，org.reactivestreams.Subscriber 没有添加资源、取消资源或者从外部进行请求的语义。

为了弥合这一差异，我们为 Flowable（和 Observable）定义了抽象类 DefaultSubscriber、ResourceSubscriber 和 DisposableSubscriber（以及 XObserver 变种形式），提供了像 rx.Subscriber 一样的资源跟踪（以 Disposable 的形式）功能，并且能够在外部通过 dispose() 进行取消。

```
ResourceSubscriber<Integer> subscriber = new ResourceSubscriber<Integer>() {
    @Override
    public void onStart() {
        request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(Integer t) {
        System.out.println(t);
    }

    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Done");
    }
};

Flowable.range(1, 10).delay(1, TimeUnit.SECONDS).subscribe(subscriber);

subscriber.dispose();
```

注意，同样因为反应式流的兼容性，onCompleted 重命名为 onComplete，去掉了最后结尾的字母 d。

在 1.*x* 中，Observable.subscribe(Subscriber) 返回的是 Subscription，所以用户经常会将 Subscription 添加到一个 CompositeSubscription 中，如下所示。

```
CompositeSubscription composite = new CompositeSubscription();

composite.add(Observable.range(1, 5).subscribe(new TestSubscriber<Integer>()));
```

根据反应式流规范，`Publisher.subscribe` 返回的是 `void`，所以这种模式本身不再适用于 2.0。为了修正这一点，我们在每个反应式基础类中都添加了 `E subscribeWith(E subscriber)` 方法，它会原样返回传入的 `subscriber/observer`。在前面两个样例中，2.*x* 的代码可以如下所示，因为 `ResourceSubscriber` 直接实现了 `Disposable`。

```
CompositeDisposable composite2 = new CompositeDisposable();

composite2.add(Flowable.range(1, 5).subscribeWith(subscriber));
```

从onSubscribe/onStart调用请求

由于请求管理的运行机制，在 `Subscriber.onSubscribe` 或 `ResourceSubscriber.onStart` 中调用 `request(n)` 可能会立即触发对 `onNext` 的调用，对 `onNext` 的调用有可能会在 `request()` 返回至 `onSubscribe/onStart` 方法前执行。

```
Flowable.range(1, 3).subscribe(new Subscriber<Integer>() {

    @Override
    public void onSubscribe(Subscription s) {
        System.out.println("OnSubscribe start");
        s.request(Long.MAX_VALUE);
        System.out.println("OnSubscribe end");
    }

    @Override
    public void onNext(Integer v) {
        System.out.println(v);
    }

    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Done");
    }
});
```

上述代码会打印出以下结果。

```
OnSubscribe start
1
2
3
Done
OnSubscribe end
```

这里的问题在于，如果有人希望在调用 `request` 之后在 `onSubscribe/onStart` 中执行一些初

始化逻辑，那么 onNext 有可能会看到初始化的效果，也有可能看不到。为了避免这种情况，需要确保在 onSubscribe/onStart 中，等到所有的初始化都完成**后**再调用 request。

这种行为与 1.*x* 不同，在 1.*x* 中，request 会经过一个延迟的逻辑来累积请求，直到上游的 Producer 在某个时刻出现（这种行为给所有 1.*x* 中的操作符和消费者带来了一定的开销）。在 2.*x* 中，始终会首先得到一个 Subscription，在 90% 的情况下没有必要对请求进行延迟。

C.1.13 Subscription

在 RxJava 1.*x* 中，rx.Subscription 接口负责流和资源的生命周期管理，即取消对序列的订阅并释放通用的资源，如调度的任务。反应式流规范使用这个名称来指定源和消费者的交互点：org.reactivestreams.Subscription。它允许请求上游流中的一个正数，并允许取消对序列的订阅。

为了避免名称的冲突，1.*x* 的 rx.Subscription 重命名为 io.reactivex.Disposable（有点类似于 .NET 自己的 IDisposable）。

因为反应式流规范的基础接口 org.reactivestreams.Publisher 将 subscribe() 方法定义为 void，所以 Flowable.subscribe(Subscriber) 不会再返回任何 Subscription（或 Disposable）。其他的反应式基础类型也遵循这个签名，分别对应各自的订阅者类型。

原始的 Subscription 类型分别进行了重命名和更新。

- CompositeSubscription 变为 CompositeDisposable。
- SerialSubscription 和 MultipleAssignmentSubscription 合并为 SerialDisposable。set() 方法取消了旧值，但是 replace() 方法不会这样做。
- RefCountSubscription 被移除。

C.1.14 回压

反应式流规范要求操作符支持回压，尤其是消费者没有请求数据的时候，要确保它们不会溢出。新的基础反应式类型 Flowable 的操作符能够正确地考虑下游请求的数量，但是这并不意味着 MissingBackpressureException 不会再出现。这个异常依然存在，但是这次，不能发布更多 onNext 信号的操作符将发布这个异常（以识别哪个操作符没有恰当地处理回压）。

作为替代方案，2.*x* 的 Observable 完全不支持回压，可以作为切换的备用选项。

C.1.15 遵循反应式流协议

更新至 2.0.7 版本

在 2.0.7 版本中，基于 Flowable 的源和操作符已经完全符合 1.0.0 版本的反应式流规范。

在 2.0.7 版本之前，为了实现同等级别的兼容性，必须要使用 strict() 操作符。在 2.0.7 版本中，strict() 操作符会返回 this，前者已经被弃用，在 2.1.0 版本中将完全移除该操作符。

作为 RxJava 2.0 版本的主要目标之一，设计主要聚焦于性能方面。为了实现该目标，RxJava 2.0.7 添加了一个自定义的 io.reactivex.FlowableSubscriber 接口（扩展了

org.reactivestreams.Subscriber)，但是并没有为其添加新方法。这个新的接口**限制在 RxJava 2.0 中**使用，它代表了 Flowable 的某个消费者在反应式规范 1.0.0 版本的 §1.3、§2.3、§2.12 和 §3.9 方面要放松限制。

- §1.3 规则放宽：如果 FlowableSubscriber 在 onSubscribe 中调用 request()，onSubscribe 可能会与 onNext 并发运行，FlowableSubscriber 需要确保 onSubscribe 中的剩余指令与 onNext 之间的线程安全性。
- §2.3 规则放宽：在 FlowableSubscriber.onComplete() 或 FlowableSubscriber.onError() 中调用 Subscription.cancel 和 Subscription.request 视为不进行任何操作。
- §2.12 规则放宽：如果相同的 FlowableSubscriber 实例被订阅到了多个源上，它必须要确保其 onXXX 方法保持线程安全。
- §3.9 规则放宽：发出非正数的 request() 不会停止当前的流，而是会通过 RxJavaPlugins.onError 发布错误信号。

从用户的角度，如果使用 subscribe 而不是 Flowable.subscribe(Subscriber<? super T>)，则无须对这个变更采取任何措施，也不会有额外的惩罚。

如果用户组合使用了 Flowable.subscribe(Subscriber<? super T>) 和内置的 RxJava 实现，比如 DisposableSubscriber、TestSubscriber 和 ResourceSubscriber，而代码没有针对 2.0.7 重新编译，那么这会带来一个很小的运行时开销（一个 instanceof 检查）。

如果之前已经有了自定义的 Subscriber 类实现，那么使用它订阅 Flowable 会添加一个内部包装器，以确保观察的 Flowable 能够 100% 遵循规范，但是会给每个条目带来一定的开销。

为了移除这些额外的开销，Flowable.subscribe(FlowableSubscriber<? super T>) 添加了一个新方法，它能够暴露 2.0.7 版本之前的原始行为。建议新的自定义消费者实现 FlowableSubscriber，而不仅仅是实现 Subscriber。

C.1.16 运行时挂钩

2.*x* 重新设计了 RxJavaPlugins 类，从而支持在运行时修改挂钩。测试想要覆盖调度器和基础反应式类型的生命周期，我们可以通过回调函数按不同情况实现。

基于 RxJavaObservableHook 的相关类已经被移除，RxJavaHooks 的功能合并到了 RxJavaPlugins 中。

C.1.17 错误处理

2.*x* 版本有一个很重要的设计需求：不吞噬任何 Throwable 错误。这意味着某些无法发布的错误需要得到特殊的处理，因为下游的生命周期已经到了终结状态，或者下游取消了对一个即将发布错误的序列的订阅。

这样的错误会被路由至 RxJavaPlugins.onError 处理器。这个处理器可以通过 RxJavaPlugins.setErrorHandler(Consumer<Throwable>) 进行重写。如果没有指定处理器，RxJava 会默认在控制台打印出 Throwable 的堆栈，并调用当前线程的未捕获异常处理代码（uncaught exception handler）。

在桌面 Java 上，后面提到的处理代码不会在 ExecutorService 支撑的 Scheduler 做任何事情，应用程序会继续运行。但是，Android 会更加严格，如果遇到这样的未捕获异常，将会终止应用程序。

如果这种行为是必要的，那另当别论。但是在有些场景下，如果我们想要避免对未捕获异常处理代码的调用，那么使用 RxJava 2.0（直接或间接）的**最终应用程序**需要定义一个不执行任何操作（no-op）的处理代码。

```
//如果支持Java 8 lambda表达式的话
RxJavaPlugins.setErrorHandler(e -> { });

//如果没有Retrolambda或Jack的话
RxJavaPlugins.setErrorHandler(Functions.<Throwable>emptyConsumer());
```

不建议中间库在自己的测试环境之外更改错误处理代码。

令人遗憾的是，RxJava 无法判断哪些异常超出了生命周期，未投递的异常是否应该终止应用程序。识别这些异常的来源非常麻烦，如果它们来源于链中的某个较为底层的源并路由至 RxJavaPlugins.onError，这会尤为困难。

因此，2.0.6 版本引入了特定的异常包装器，帮助识别和跟踪当错误出现时的情况。

- OnErrorNotImplementedException：重新引入该异常以探测用户忘记为 subscribe() 添加错误处理的场景。
- ProtocolViolationException：表明操作符中的缺陷。
- UndeliverableException：包装由于 Subscriber/Observer 的生命周期限制无法进行投递的原始异常。RxJavaPlugins.onError 会自动应用该异常，并且带有完整的堆栈轨迹，这样有助于发现哪个操作符重新路由了原始的错误。

如果未投递的异常是 NullPointerException、IllegalStateException（UndeliverableException 和 ProtocolViolationException 扩展了该异常）、IllegalArgumentException、CompositeException、MissingBackpressureException 或 OnErrorNotImplementedException 的实例或后代，将不会包装为 UndeliverableException。

此外，有些第三方库被 cancel/dispose 打断的时候，大多数情况下抛出的异常会无法投递。2.0.6 版本有一些内部变化，它会在处理任务或 worker（这会导致目标线程的中断）之前处理 Subscription/Disposable。

```
//在某些库中
try {
    doSomethingBlockingly()
} catch (InterruptedException ex) {
    //检查是否因为取消而中断
    //如果是这样的话，则没有必要发布 InterruptedException信号
    if (!disposable.isDisposed()) {
        observer.onError(ex);
    }
}
```

如果库 / 代码已经这样做了，未投递的 InterruptedException 应该到此为止。如果以前没

有采取这种模式，我们推荐更新相关的代码 / 库。

如果决定添加非空的全局错误消费者，它会管理较为典型的未投递异常，处理过程会判断这是一个 bug 还是一种可忽略的应用程序 / 网络状态，如下所示。

```
RxJavaPlugins.setErrorHandler(e -> {
    if (e instanceof UndeliverableException) {
        e = e.getCause();
    }
    if ((e instanceof IOException) || (e instanceof SocketException)) {
        //不相关的网络问题或API导致了取消
        return;
    }
    if (e instanceof InterruptedException) {
        //一些阻塞代码被dispose调用中断
        return;
    }
    if ((e instanceof NullPointerException) || (e instanceof
    IllegalArgumentException)) {
        //似乎是应用的bug
        Thread.currentThread().getUncaughtExceptionHandler()
            .handleException(Thread.currentThread(), e);
        return;
    }
    if (e instanceof IllegalStateException) {
        //RxJava或自定义操作符中的bug
        Thread.currentThread().getUncaughtExceptionHandler()
            .handleException(Thread.currentThread(), e);
        return;
    }
    Log.warning("Undeliverable exception received, not sure what to do", e);
});
```

C.1.18 Scheduler

2.*x* 版本的 API 依然支持大多数的默认调度器类型：`computation`、`io`、`newThread` 和 `trampoline`，它们可以通过 `io.reactivex.schedulers.Schedulers` 工具类来使用。

2.*x* 中已经没有 `immediate` 了，它经常被误用，并且没有正确地实现 `Scheduler` 规范。它为延迟操作包含了阻塞式的休眠，并且不支持递归调度。现在，应该使用 `Schedulers.trampoline()` 来替代它。

`Schedulers.test()` 也被移除了，从而避免出现与其他默认调度器概念上的混淆。其他的类型都会返回一个“全局”调度器，而 `test()` 始终都会返回一个新的 `TestScheduler` 实例。我们鼓励开发人员在代码中直接使用 `new TestScheduler()`。

`io.reactivex.Scheduler` 抽象基础类现在能够直接调度任务，不再需要创建和销毁 `Worker`（经常被遗忘）了。

```
public abstract class Scheduler {

    public Disposable scheduleDirect(Runnable task) { ... }
```

```
    public Disposable scheduleDirect(Runnable task, long delay, TimeUnit unit) {
... }

    public Disposable scheduleDirectPeriodically(Runnable task, long initialDelay,
        long period, TimeUnit unit) { ... }

    public long now(TimeUnit unit) { ... }

    //……其余都是一样的：生命周期方法、创建worker
}
```

主要目的是避免一次性的任务跟踪 `Worker` 带来的开销。方法有一个默认实现，它会重用 `createWorker`，如果必要，可以用更高效的实现来重写它。

方法会返回调度器本身对当前时间的理解，`now()` 改为接收一个 `TimeUnit`，用来指明度量单位。

C.1.19　进入反应式的世界

RxJava 1.*x* 的一个设计缺陷是对外暴露了 `rx.Observable.create()` 方法，尽管它非常强大，但它并不是进入反应式世界应该使用的典型操作符。令人遗憾的是，太多的人依赖这个方法，我们不能移除它或者对其重新命名。

2.*x* 版本是一个全新的开始，我们不会再犯这样的错误。每个反应式基础类型 `Flowable`、`Observable`、`Single`、`Maybe` 和 `Completable` 都有一个安全的 `create` 操作符，在回压（针对 `Flowable`）和取消（针对所有类型）方面，它们都能完成正确的操作。

```
Flowable.create((FlowableEmitter<Integer> emitter) -> {
    emitter.onNext(1);
    emitter.onNext(2);
    emitter.onComplete();
}, BackpressureStrategy.BUFFER);
```

实际上，1.*x* 的 `fromEmitter`（以前被称为 `fromAsync`）已经更名为 `Flowable.create`。其他反应式基础类型有类似的 `create` 方法（缺少回压策略）。

C.1.20　脱离反应式的世界

除了使用对应的消费者（`Subscriber`、`Observer`、`SingleObserver`、`MaybeObserver` 和 `CompletableObserver`）和基于函数式接口（比如 `subscribe(Consumer<T>, Consumer <Throwable>, Action)` 的消费者订阅基础类型，1.*x* 中独立的 `BlockingObservable`（以及相似的类）已经集成到主要的反应式类型。现在，我们可以通过 `blockingX` 操作直接阻塞以获取结果。

```
List<Integer> list = Flowable.range(1, 100).toList().blockingGet();
//toList()
返回Single
Integer i = Flowable.range(100, 100).blockingLast();
```

（原因有两个：使用该库作为同步 Java 8 Streams 类处理代码的性能和易用性。）

rx.Subscriber 和 org.reactivestreams.Subscriber 的另一个明显差异是：在 2.*x* 中，Subscriber 和 Observer 只会抛出致命的异常（参见 Exceptions.throwIfFatal()），而不抛出其他异常（反应式流规范能够在 onSubscribe、onNext 或 onError 接收到 null 时抛出 NullPointerException，但是 RxJava 根本不允许 null 出现）。这意味着如下代码是非法的。

```
Subscriber<Integer> subscriber = new Subscriber<Integer>() {
    @Override
    public void onSubscribe(Subscription s) {
        s.request(Long.MAX_VALUE);
    }

    public void onNext(Integer t) {
        if (t == 1) {
            throw new IllegalArgumentException();
        }
    }

    public void onError(Throwable e) {
        if (e instanceof IllegalArgumentException) {
            throw new UnsupportedOperationException();
        }
    }

    public void onComplete() {
        throw new NoSuchElementException();
    }
};

Flowable.just(1).subscribe(subscriber);
```

这同样适用于 Observer、SingleObserver、MaybeObserver 和 CompletableObserver。

鉴于很多面向 1.*x* 的代码没有按照该要求来编写，因此我们引入了 safeSubscribe 方法，来处理这些不符合要求的消费者。

另一种替代方案是使用 subscribe(Consumer<T>, Consumer<Throwable>, Action)（和类似的方法），以提供可以抛出异常的回调 / lambda 表达式。

```
Flowable.just(1)
.subscribe(
    subscriber::onNext,
    subscriber::onError,
    subscriber::onComplete,
    subscriber::onSubscribe
);
```

C.1.21 测试

在测试方面，RxJava 2.*x* 的运行方式与 1.*x* 相同。Flowable 可以使用 io.reactivex.subscribers.TestSubscriber 进行测试，而不支持回压的 Observable、Single、Maybe 和 Completable 可以使用 io.reactivex.observers.TestObserver。

1. test()操作符

为了支撑我们的内部测试，现在所有的反应式基础类型都有 test() 方法（这带来了巨大的便利），会返回 TestSubscriber 或 TestObserver。

```
TestSubscriber<Integer> ts = Flowable.range(1, 5).test();

TestObserver<Integer> to = Observable.range(1, 5).test();

TestObserver<Integer> tso = Single.just(1).test();

TestObserver<Integer> tmo = Maybe.just(1).test();

TestObserver<Integer> tco = Completable.complete().test();
```

第二个便利之处在于大多数的 TestSubscriber/TestObserver 方法都会返回实例本身，可以将各种 assertX 方法链接起来。第三个便利之处是我们可以流畅地测试源，不必在代码中创建或引入 TestSubscriber/TestObserver 实例。

```
Flowable.range(1, 5)
.test()
.assertResult(1, 2, 3, 4, 5)
;
```

❑ 值得注意的新断言方法

- assertResult(T... items)：断言如果进行订阅，则按照给定的顺序依次接收给定的条目，随后接收到 onComplete 通知，没有错误出现。
- assertFailure(Class<? extends Throwable> clazz, T... items)：断言如果进行订阅，则按照给定的顺序依次接收给定的条目，随后出现一个 Throwable 错误，使用 clazz.isInstance() 判断返回 true。
- assertFailureAndMessage(Class<? extends Throwable> clazz, String message, T... items)：与 assertFailure 相同，除此之外，还要校验 getMessage() 包含特定的信息。
- awaitDone(long time, TimeUnit unit)：等待终端事件（阻塞式），如果出现超时，取消对序列的订阅。
- assertOf(Consumer<TestSubscriber<T>> consumer)：将一些断言组合到一个流畅的链中（在内部用于融合测试，因为现在操作符融合不是公共 API 的一部分）。

这里将 Flowable 替换为 Observable 的一个益处是：由于从 TestSubscriber 到 TestObserver 的隐式类型转换，测试代码部分完全不需要任何变化。

2. 预先取消和请求

TestObserver 的 test() 方法有一个 test(boolean cancel) 的重载形式，它会在真正订阅之前就取消 TestSubscriber/TestObserver。

```
PublishSubject<Integer> pp = PublishSubject.create();

//还没有人订阅
assertFalse(pp.hasSubscribers());

pp.test(true);
```

```
//没有人能够保持订阅
assertFalse(pp.hasSubscribers());
```

`TestSubscriber` 具有 `test(long initialRequest)` 和 `test(long initialRequest, boolean cancel)` 的重载形式，它能够指定初始的请求数量以及 `TestSubscriber` 是否应该立即取消。如果指定了 `initialRequest`，还可以使用 `TestSubscriber` 提供的 `requestMore(long)`，从而能够以连贯的方式保持请求。

```
Flowable.range(1, 5)
.test(0)
.assertValues()
.requestMore(1)
.assertValues(1)
.requestMore(2)
.assertValues(1, 2, 3)
.requestMore(2)
.assertResult(1, 2, 3, 4, 5);
```

而作为替代方案，必须要捕获 `TestSubscriber` 实例，以访问其 `request()` 方法。

```
PublishProcessor<Integer> pp = PublishProcessor.create();

TestSubscriber<Integer> ts = pp.test(0L);

ts.request(1);

pp.onNext(1);
pp.onNext(2);

ts.assertFailure(MissingBackpressureException.class, 1);
```

3. 测试异步源

对于异步源，我们现在可以连贯地阻塞等待终端事件。

```
Flowable.just(1)
.subscribeOn(Schedulers.single())
.test()
.awaitDone(5, TimeUnit.SECONDS)
.assertResult(1);
```

4. Mockito & TestSubscriber

如果在 1.*x* 版本中使用 Mockito 和仿造的 `Observer`，那么现在必须要仿造 `Subscriber.onSubscribe` 方法，以发起初始请求；否则，序列将会挂起，或因 hot 类型的源出现失败。

```
@SuppressWarnings("unchecked")
public static <T> Subscriber<T> mockSubscriber() {
    Subscriber<T> w = mock(Subscriber.class);

    Mockito.doAnswer(new Answer<Object>() {
        @Override
        public Object answer(InvocationOnMock a) throws Throwable {
            Subscription s = a.getArgumentAt(0, Subscription.class);
            s.request(Long.MAX_VALUE);
```

```
                return null;
            }
        }).when(w).onSubscribe((Subscription)any());

        return w;
    }
```

C.1.22　操作符的差异

在 2.*x* 中，大多数操作符都保留了下来，并且行为与 1.*x* 中相同。下面的子章节列出了每个反应式基础类型以及 1.*x* 和 2.*x* 之间的差异。

很多操作符都增加了重载版本，可以指定内部缓冲的大小或者在上游流（或内部源）上预先抓取的数量。

有些操作符的重载形式使用特定的后缀进行了重命名，比如 `fromArray`、`fromIterable` 等。原因是库针对 Java 8 进行编译时，javac 通常无法消除函数式接口类型之前的歧义。

在 1.*x* 版本中使用 `@Beta` 或 `@Experimental` 标注的操作符现在提升到了标准水平。

从1.*x*的`Observable`到2.*x*的`Flowable`

- 工厂方法（见表 C-1）

表C-1：工厂方法

1.*x*	2.*x*
`amb`	添加 `amb(ObservableSource...)` 重载形式，废弃 2~9 个参数的版本
`RxRingBuffer.SIZE`	`bufferSize()`
`combineLatest`	添加可变参数（varargs）的重载形式，添加带有 `bufferSize` 参数的重载形式，废弃 `combineLatest(List)`
`concat`	添加 `prefetch` 参数的重载版本，废弃 5~9 个源的重载版本，使用 `concatArray` 作为替代方案
N/A	添加 `concatArray` 和 `concatArrayDelayError`
N/A	添加 `concatArray` 和 `concatArrayDelayError`
`concatDelayError`	添加重载形式，能够延迟到当前源结束或所有源结束
`concatEagerDelayError`	添加重载形式，能够延迟到当前源结束或所有源结束
`create(SyncOnSubscribe)`	替换为 `generate` 及重载形式（不同的接口，可以一次性实现它们）
`create(AsnycOnSubscribe)`	不复存在了
`create(OnSubscribe)`	通过安全的 `create(FlowableOnSubscribe, BackpressureStrategy)` 重新进行了定义，通过 `unsafeCreate()` 支持原有功能
`from`	为消除歧义，拆分到了 `fromArray`、`fromIterable` 和 `fromFuture` 中
N/A	添加 `fromPublisher`
`fromAsync`	重命名为 `create()`
N/A	添加 `intervalRange()`
`limit`	废弃，使用 `take`
`merge`	添加带有 `prefetch` 的重载形式

（续）

1.*x*	2.*x*
mergeDelayError	添加带有 prefetch 的重载形式
sequenceEqual	添加带有 bufferSize 的重载形式
switchOnNext	添加带有 prefetch 的重载形式
switchOnNextDelayError	添加带有 prefetch 的重载形式
timer	废弃过时的重载版本
zip	添加带有 bufferSize 和 delayErrors 功能的重载版本，为了消除歧义改为 zipArray 和 zipIterable

- **实例方法（见表 C-2）**

表C-2：实例方法

1.*x*	2.*x*
all	RC3 返回 Single<Boolean>
any	RC3 返回 Single<Boolean>
asObservable	重命名为 hide()，会隐藏所有的标识
buffer	添加使用自定义 Collection 供应者的重载版本
cache(int)	废弃
collect	RC3 返回 Single<U>
collect(U, Action2<U, T>)	为了消除歧义改为 collectInto，并且 RC3 返回 Single<U>
concatMap	添加带有 prefetch 的重载形式
concatMapDelayError	添加带有 prefetch 的重载形式，能够延迟到当前源结束或延迟到所有源结束
concatMapEager	添加带有 prefetch 的重载形式
concatMapEagerDelayError	添加了带有 prefetch 的重载形式，能够延迟到当前源结束或所有源结束
count	RC3 返回 Single<Long>
countLong	废弃，使用 count 替代
distinct	添加使用自定义 Collection 供应者的重载版本
doOnCompleted	重命名为 doOnComplete，注意少了一个 d
doOnUnsubscribe	重命名为 Flowable.doOnCancel，其他类型重命名为 doOnDispose
N/A	添加了 doOnLifecylce 来处理窥探 onSubscribe、request 和 cancel 的要求
elementAt(int)	如果源的长度小于索引，RC3 将不再标志 NoSuchElementException
elementAt(Func1, int)	废弃，使用 filter(predicate).elementAt(int)
elementAtOrDefault(int, T)	重命名为 elementAt(int, T)，RC3 返回 Single<T>
elementAtOrDefault(Func1, int, T)	废弃，使用 filter(predicate).elementAt(int, T)
first()	RC3 重命名为 firstElement 并返回 Maybe<T>
first(Func1)	废弃，使用 filter(predicate).first()
firstOrDefault(T)	重命名为 first(T)，RC3 返回 Single<T>

（续）

1.*x*	2.*x*
firstOrDefault(Func1, T)	废弃，使用 filter(predicate).first(T)
flatMap	添加带有 prefetch 的重载形式
N/A	添加 forEachWhile(Predicate<T>, [Consumer<Throwable>, [Action]])，以便于条件性地停止消费事件
groupBy	添加使用 bufferSize 和 delayError 选项的重载版本，自定义的内部映射版本未列入 RC1
ignoreElements	RC3 返回 Completable
isEmpty	RC3 返回 Single<Boolean>
last()	RC3 重命名为 lastElement 并返回 Maybe<T>
last(Func1)	废弃，重命名为 filter(predicate).last()
lastOrDefault(T)	重命名为 last(T)，RC3 返回 Single<T>
lastOrDefault(Func1, T)	废弃，使用 filter(predicate).last(T)
nest	废弃，使用手动 just
publish(Func1)	添加带有 prefetch 的重载形式
reduce(Func2)	RC3 返回 Maybe<T>
N/A	添加 reduceWith(Callable, BiFunction)，以单个 Subscriber 的方式减少，返回 Single<T>
N/A	添加 repeatUntil(BooleanSupplier)
repeatWhen(Func1, Scheduler)	废弃该重载形式，使用 subscribeOn(Scheduler).repeatWhen (Function) 替代
retry	添加 retry(Predicate)、retry(int, Predicate)
N/A	添加 retryUntil(BooleanSupplier)
retryWhen(Func1, Scheduler)	废弃该重载形式，使用 subscribeOn(Scheduler).retryWhen (Function) 替代
sample	如果上游流在这个时间段内结束，不会发布最后一个条目，添加带有 emitLast 参数的重载形式
N/A	添加 scanWith(Callable, BiFunction)，基于单个 Subscriber 的方式进行扫描
single()	RC3 重命名为 singleElement，返回 Maybe<T>
single(Func1)	废弃，使用 filter(predicate).single()
singleOrDefault(T)	重命名为 single(T)，RC3 返回 Single<T>
singleOrDefault(Func1, T)	废弃，使用 filter(predicate).single(T)
skipLast	添加带有 bufferSize 和 delayError 选项的重载版本
startWith	具有 2~9 个参数的重载形式已废弃，使用 startWithArray 作为替代方案
N/A	为了消除歧义性，新增了 startWithArray
subscribe	不再使用安全的包装器包装所有消费者类型（如 Observer），就像 1.*x* 的 unsafeSubscribe 已经不可用了。显式使用 safeSubscribe 获取消费者类型的安全包装器

（续）

1.*x*	2.*x*
N/A	添加 subscribeWith，它会在订阅之后返回它的输入
switchMap	添加带有 prefetch 的重载形式
switchMapDelayError	添加带有 prefetch 的重载形式
takeLastBuffer	废弃
N/A	添加 test()（返回订阅它的 TestSubscriber），具有重载形式以便于流畅地测试
throttleLast	如果在这段时间内上游流结束，不会发布最后一个条目，请使用带有 emitLast 参数的 sample
timeout(Func0<Observable>, ...)	签名更改为 timeout(Publisher, ...)，废弃了函数。如果必要的话，请使用 defer(Callable<Publisher>>)
toBlocking().y	内联为 blockingY() 操作符，toFuture 除外
toCompletable	RC3 已废弃，使用 ignoreElements
toList	RC3 返回 Single<List<T>>
toMap	RC3 返回 Single<Map<K, V>>
toMultimap	RC3 返回 Single<Map<K, Collection<V>>>
N/A	添加 toFuture
N/A	添加 toObservable
toSingle	RC3 已废弃，使用 single(T)
toSortedList	RC3 返回 Single<List<T>>
unsafeSubscribe	已移除，因为反应式规范要求 onXXX 方法不能出现崩溃，所以默认情况下在 subscribe 中没有安全网的保护。新增的 safeSubscribe 方法会围绕消费者类型显式添加一个安全包装器
withLatestFrom	废弃 5~9 个源的重载形式
zipWith	添加带有 prefetch 和 delayErrors 选项的重载形式

- **各种返回类型（见表 C-3）**

在 2.*x* 中，有些只生成一个值或者会产生错误的操作符现在会返回 Single（如果允许空的事件源，那么会返回 Maybe）。

（注意，这在 RC2 和 RC3 中是“实验性的”，目的是了解使用这种混合类型序列编程的感觉，以及是否需要太多 toObservable/toFlowable 的反向转换。）

表C-3：各种返回类型

操　作　符	旧的返回类型	新的返回类型	备　　注
all(Predicate)	Observable<Boolean>	Single<Boolean>	如果所有元素均匹配断言，发布 true
any(Predicate)	Observable<Boolean>	Single<Boolean>	如果任意一个元素均匹配断言，发布 true
count()	Observable<Long>	Single<Long>	统计序列中元素的数量

（续）

操　作　符	旧的返回类型	新的返回类型	备　　注
`elementAt(int)`	`Observable<T>`	`Maybe<T>`	发布给定索引的元素，或者直接完成
`elementAt(int, T)`	`Observable<T>`	`Single<T>`	发布给定索引的元素或默认值
`elementAtOrError(int)`	`Observable<T>`	`Single<T>`	发布给定索引的元素或 `NoSuchElementException`
`first(T)`	`Observable<T>`	`Single<T>`	发布第一个元素或 `NoSuchElementException`
`firstElement()`	`Observable<T>`	`Maybe<T>`	发布第一个元素，或者直接完成
`firstOrError()`	`Observable<T>`	`Single<T>`	发布第一个元素，如果源为空，抛出 `NoSuchElementException`
`ignoreElements()`	`Observable<T>`	`Completable`	忽略除终端事件之外的所有内容
`isEmpty()`	`Observable<Boolean>`	`Single<Boolean>`	如果源为空，发布 true
`last(T)`	`Observable<T>`	`Single<T>`	发布最后一个元素或默认值
`lastElement()`	`Observable<T>`	`Maybe<T>`	发布最后一个元素或直接完成
`lastOrError()`	`Observable<T>`	`Single<T>`	发布最后一个元素，如果源为空，抛出 `NoSuchElementException`
`reduce(BiFunction)`	`Observable<T>`	`Maybe<T>`	发布约减后的值或直接完成
`reduce(Callable, BiFunction)`	`Observable<U>`	`Single<U>`	发布约减后的值（或初始值）
`reduceWith(U, BiFunction)`	`Observable<U>`	`Single<U>`	发布约减后的值（或初始值）
`single(T)`	`Observable<T>`	`Single<T>`	返回唯一的元素或默认值
`singleElement()`	`Observable<T>`	`Maybe<T>`	返回唯一的元素或直接完成
`singleOrError()`	`Observable<T>`	`Single<T>`	返回有且仅有的一个元素，如果源的长度超过 1 个条目，抛出 `IndexOutOfBounds-Exception`；如果源为空，抛出 `NoSuchElementException`
`toList()`	`Observable<List<T>>`	`Single<List<T>>`	将所有的元素收集到一个 `List` 中
`toMap()`	`Observable<Map<K, V>>`	`Single<Map<K, V>>`	将所有的元素收集到一个 `Map` 中
`toMultimap()`	`Observable<Map<K, Collection<V>>>`	`Single<Map<K, Collection<V>>>`	将所有的元素收集到一个带有集合的 `Map` 中
`toSortedList()`	`Observable<List<T>>`	`Single<List<T>>`	将所有的元素收集到一个 `List` 中并排序

❑ 移除

为了确保 2.0 版本的最终 API 尽可能地简洁，我们移除了候选发布版本之间的方法和其他组件，而不是反对他们（见表 C-4）。

表C-4：移除与替代

在版本中移除	组　　件	备　　注
RC3	`Flowable.toCompletable()`	使用 `Flowable.toCompletable()`
RC3	`Flowable.toSingle()`	使用 `Flowable.single(T)`
RC3	`Flowable.toMaybe()`	使用 `Flowable.single(T)`
RC3	`Observable.toCompletable()`	使用 `Observable.ignoreElements()`
RC3	`Observable.toSingle()`	使用 `Observable.toSingle()`
RC3	`Observable.toMaybe()`	使用 `Observable.singleElement()`

C.1.23　杂项变更

doOnCancel/doOnDispose/unsubscribeOn

在 1.*x* 中遇到终端事件时，doOnUnsubscribe 始终都会执行，因为 1.*x* 的 SafeSubscriber 本身就会调用 unsubscribe。这实际上是没有必要的。反应式规范规定如果某个终端事件抵达 Subscriber，上游 Subscription 应该视为已经取消，因此调用 cancel() 就没有什么意义了。

基于相同的原因，在常规的终端路径中，unsubscribeOn 不会被调用，只有实际调用 cancel（或 dispose）的时候，才会调用这个链。

因此，如下序列不会调用 doOnCancel。

```
Flowable.just(1, 2, 3)
.doOnCancel(() -> System.out.println("Cancelled!"))
.subscribe(System.out::println);
```

但是，如下代码会调用 doOnCancel，因为 take 操作符在指定数量的 onNext 事件被投递之后会取消订阅。

```
Flowable.just(1, 2, 3)
.doOnCancel(() -> System.out.println("Cancelled!"))
.take(2)
.subscribe(System.out::println);
```

如果需要在正常的终端或取消场景下执行一些清理操作，那么可以考虑使用 using 作为替代方案。

另外，还可以考虑使用 doFinally 操作符（2.0.1 版本引入，2.1 版本中进行了标准化）。它会在源完成的时候调用开发人员指定的 Action，无论是因为错误而失败还是因为执行了 cancel/dispose。

```
Flowable.just(1, 2, 3)
.doFinally(() -> System.out.println("Finally"))
.subscribe(System.out::println);

Flowable.just(1, 2, 3)
.doFinally(() -> System.out.println("Finally"))
.take(2) //在2个元素之后消去上面的项
.subscribe(System.out::println);
```

C.2 RxJava 2.0版本中的回压

C.2.1 回压概述

回压是指在 Flowable 处理管道上，一些异步的处理阶段无法快速地处理值，因此需要有一种方式来通知上游生产者放慢速度。

需要回压功能的典型场景（生产者为 hot 类型的源时）如下所示。

```
PublishProcessor<Integer> source = PublishProcessor.create();

source
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}

Thread.sleep(10_000);
```

在本例中，主线程会为终端消费者生成 100 万个条目，消费者会在后台线程中对它们进行处理。compute(int) 方法可能会耗费一些时间，Flowable 操作符链可能也会添加用于条目处理的时间。但是，具有 for 循环的生成线程并不知道这一点，会不断调用 onNext。

在内部，异步操作符有一个缓冲区来存放这样的元素，直到它们得到处理。在经典的 Rx.NET 和早期的 RxJava 中，这些缓冲区没有边界限制，即在本例中会存放近 100 万个条目。如果在程序中有 10 亿个元素或相同的 100 万个序列出现了 1000 次，那么问题就出现了，这会导致 OutOfMemoryError，通常还会因为过多的 GC 开销导致系统变慢。

在 RxJava 中，错误处理是一等公民，并且会使用操作符对其进行处理（通过 onErrorXXX）。与之类似，回压是开发人员必须要考虑的数据流另一个属性（通过 onBackpressureXXX 操作符）。

除了上面提到的 PublishProcessorabove，还有其他操作符不支持回压，这大多数是功能方面的原因。例如，操作符 interval 会周期性地发布值，如果对其实施回压，会导致时间段相对于墙上时钟（wall clock）产生偏差。

在现代 RxJava 中，大多数异步操作符都有一个有界的内部缓冲，比如上面提到的 observeOn。如果尝试溢出这个缓冲区，整个序列将会终止，并且会抛出 MissingBackpressureException。每个操作符的文档都有关于回压行为的描述。

但是，在常规 cold 类型的序列中，回压的行为更微妙一些（它不会也不应该产生 MissingBackpressureException）。如果将第一个样例重写为以下代码。

```
Flowable.range(1, 1_000_000)
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

Thread.sleep(10_000);
```

这里不会出现任何错误，程序能够以很少的内存占用平稳运行。原因在于很多源操作符能够按照需求“生成”值。因此，observeOn 操作符告诉 range 最多要生成多少值，才能确保 observeOn 缓冲区可以立刻容纳而不出现溢出。

这个协商的过程基于计算机科学中的协程概念（co-routine，我调用你，你调用我）。操作符 range 发送一个回调给 observeOn，这个回调的表现形式是 org.reactivestreams.Subscription 接口的实现，从而能够调用它的（内部 Subscriber 的）onSubscribe。反过来，observeOn 调用 Subscription.request(n)，调用时带有一个值以告诉 range 允许生成（即调用其 onNext）的**额外**元素数量。随后，observeOn 负责在合适的时间以合适的值调用 request 方法，以确保数据正常流动，不会出现溢出的情况。

终端消费者很少需要表述回压（因为它们与直接上游同步，而回压是调用栈阻塞而自然发生的），但是这样做可以更容易地理解它的运行原理，如下所示。

```
Flowable.range(1, 1_000_000)
.subscribe(new DisposableSubscriber<Integer>() {
    @Override
    public void onStart() {
        request(1);
    }

    public void onNext(Integer v) {
        compute(v);

        request(1);
    }

    @Override
    public void onError(Throwable ex) {
        ex.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Done!");
    }
});
```

onStart 告诉 range 生成其第一个值，然后在 onNext 中接收该值。一旦 compute(int) 完成，程序就会从 range 中请求另外一个值。在 range 的原生实现中，这样的代码会递归调用 onNext，从而出现 StackOverflowError，这当然是我们不想看到的。

为了避免出现这种情况，操作符使用了所谓的蹦床逻辑（trampolining logic）来避免这种可重入调用。就 range 来说，它会记住在调用 onNext() 时有一个 request(1) 调用，一旦 onNext() 返回，它就会发起新一轮的调用，并使用下一个整数值来调用 onNext()。因此，如果两行代码互换位置，样例依然能够以相同的方式运行，如下所示。

```
@Override
    public void onNext(Integer v) {
        request(1);
```

```
            compute(v);
        }
```

但是，onStart 的情况并非如此。尽管 Flowable 的基础设施能够确保在每个 Subscriber 上最多只调用一次，对 request(1) 的调用可能会立即触发元素的发布。如果在调用 request(1) 后面有 onNext 需要的初始化逻辑，那么可能会出现异常。

```
Flowable.range(1, 1_000_000)
.subscribe(new DisposableSubscriber<Integer>() {

    String name;

    @Override
    public void onStart() {
        request(1);

        name = "RangeExample";
    }

    @Override
    public void onNext(Integer v) {
        compute(name.length + v);

        request(1);
    }

    //……剩余内容是相同的
});
```

在这个同步的场景中，执行 onStart 的时候会立即抛出 NullPointerException。如果对 request(1) 的调用触发了其他线程上对 onNext 的异步调用，在 onNext 中读取 name 与在 onStart 中写入 name 竞争公布 request，这样可能会产生更诡异的 bug。

因此，我们应该在 onStart 或更早的地方完成所有字段的初始化，最后再调用 request()。如果必要，操作符中的 request() 实现要确保正确的 happens-before 关系（即内存释放或完全隔离）。

C.2.2 onBackpressureXXX操作符

在应用程序因为 MissingBackpressureException 而失败时，大多数开发人员就会接触到回压，异常通常指向 observeOn 操作符。实际的诱因通常是使用了不支持回压的 PublishProcessor、timer() 或 interval()，或是通过 create() 创建的自定义操作符。

我们有多种方式来处理这样的场景。

1. 增加缓冲区的大小

有时候，这样的溢出是由突发的源引起的。比如，用户突然快速地点击屏幕，observeOn 在 Android 中默认的存放 16 个元素的缓冲区就会溢出。

在最近的 RxJava 版本中，大多数对回压敏感的操作符都能够让开发人员指定其内部缓冲区的大小。相关的参数通常被称为 bufferSize、prefetch 或 capacityHint。基于概述中的

样例，我们可以增加 observeOn 的大小，让它有足够的空间存放所有的值。

```
PublishProcessor<Integer> source = PublishProcessor.create();

source.observeOn(Schedulers.computation(), 1024 * 1024)
      .subscribe(e -> { }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

不过，需要注意，这只是临时的修复方法，如果源超出了预计的缓冲区大小，依然会发生溢出的情况。这时，我们可以在下面这些操作符中选择一个使用。

2. 使用标准的操作符批处理或跳过值

如果源数据能够按照批次更高效地进行处理，那么可以使用标准的批处理操作符（按照大小或时间进行批量处理），减少 MissingBackpressureException 出现的可能性。

```
PublishProcessor<Integer> source = PublishProcessor.create();

source
      .buffer(1024)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(list -> {
          list.parallelStream().map(e -> e * e).first();
      }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

如果可以安全地忽略某些值，那么我们可以使用采样（基于时间或另一个 Flowable 进行采样）或者节流操作符（throttleFirst、throttleLast、throttleWithTimeout）。

```
PublishProcessor<Integer> source = PublishProcessor.create();

source
      .sample(1, TimeUnit.MILLISECONDS)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

注意，这些操作符只是降低了下游接收值的速度，因此依然有可能会出现 MissingBackpressureException。

3. onBackpressureBuffer()

这个操作符的无参形式会在上游源和下游操作符之间重新引入一个无界的缓冲区。无界意味着只要 JVM 没有耗尽内存，它就能处理来自突发源的任意数量的事件。

```
Flowable.range(1, 1_000_000)
        .onBackpressureBuffer()
        .observeOn(Schedulers.computation(), 8)
        .subscribe(e -> { }, Throwable::printStackTrace);
```

在本例中，observeOn 的缓冲区非常小，但是不会出现 MissingBackpressureException。因为 onBackpressureBuffer 会吸收所有的 100 万个值，并将其以很小的批次传递给 observeOn。

但是需要注意，onBackpressureBuffer 以无界的形式消费它的源，这意味着它不会应用任何回压。其结果是，即使是 range 这样的回压支撑源也将完全实现。

onBackpressureBuffer 有 4 种额外的重载形式。

❑ onBackpressureBuffer(int capacity)
这是一个有界的版本，如果缓冲区达到指定容量，将发布 BufferOverflowErrorin。

```
Flowable.range(1, 1_000_000)
        .onBackpressureBuffer(16)
        .observeOn(Schedulers.computation())
        .subscribe(e -> { }, Throwable::printStackTrace);
```

越来越多的操作符可以设置缓冲区的大小，这个操作符的重要性在不断下降。其他场景中，将 onBackpressureBuffer 设置为比默认值更大的数字其实是给了操作符一个“扩展内部缓冲区”的机会。

❑ onBackpressureBuffer(int capacity, Action onOverflow)
这个重载形式会在溢出时调用一个（共享的）操作。它的用途非常有限，除了当前的调用栈之外，无法提供关于溢出的其他信息。

❑ onBackpressureBuffer(int capacity, Action onOverflow, BackpressureOverflowStrategy strategy)
这个重载形式更有用，我们可以定义在达到给定的容量时要执行什么操作。BackpressureOverflow.Strategy 实际上是一个接口，但是 BackpressureOverflow 提供了 4 个静态的域，这些域的实现代表典型的操作。

- ON_OVERFLOW_ERROR：这是前两个重载的默认行为，显示一个 BufferOverflowException。
- ON_OVERFLOW_DEFAULT：现在它和 ON_OVERFLOW_ERROR 一样。
- ON_OVERFLOW_DROP_LATEST：如果发生溢出，当前的值会直接被忽略，仅在下游请求时才会传递旧值。
- ON_OVERFLOW_DROP_OLDEST：删除缓冲区中的最旧的元素并向其中添加当前值。

```
Flowable.range(1, 1_000_000)
        .onBackpressureBuffer(16, () -> { },
            BufferOverflowStrategy.ON_OVERFLOW_DROP_OLDEST)
        .observeOn(Schedulers.computation())
        .subscribe(e -> { }, Throwable::printStackTrace);
```

注意，最后两种策略会导致流的不连续，因为它们会丢弃元素。另外，它们不会标记 BufferOverflowException。

- ❑ `onBackpressureDrop()`

下游不再准备接收值的时候，这个操作符会丢弃序列中的元素。我们可以将其视为：容量为 0 的 onBackpressureBuffer，策略为 ON_OVERFLOW_DROP_LATEST。

如果我们可以安全地忽略源中的元素（比如鼠标移动或当前 GPS 位置的信号），那么这个操作符就有了用武之地，因为随后会有更多的新值出现。

```
component.mouseMoves()
.onBackpressureDrop()
.observeOn(Schedulers.computation(), 1)
.subscribe(event -> compute(event.x, event.y));
```

它还可以与源操作符 interval() 组合起来使用。例如，如果我们想要执行一些周期性的后台任务，但是每次迭代比周期还长，此时可以安全地丢弃多余的间隔通知，因为随后会出现更多的通知。

```
Flowable.interval(1, TimeUnit.MINUTES)
.onBackpressureDrop()
.observeOn(Schedulers.io())
.doOnNext(e -> networkCall.doStuff())
.subscribe(v -> { }, Throwable::printStackTrace);
```

这个操作符有一个重载形式：onBackpressureDrop(Consumer<? super T> onDrop)。值要被丢弃的时候，会调用这个（共享的）操作。该变种形式能够让我们对值本身进行一些清理（比如释放相关的资源）。

- ❑ `onBackpressureLatest()`

最后一个操作符只会保留最后一个值，将更旧的、未投递的值覆盖。我们可以将其视为 onBackpressureBuffer 的一个变体：容量为 1，策略为 ON_OVERFLOW_DROP_OLDEST。

不同于 onBackpressureDrop，onBackpressureLatest 能够确保下游的处理速度跟不上节奏时，始终还有一个值可以被消费。在一些类似遥测的场景中，这种方式可能会非常有用，这时数据可能会以突发模式出现，但是只有最新的值才值得处理。

例如，如果用户在屏幕上点击了多次，我们只会对最后的输入做出响应。

```
component.mouseClicks()
.onBackpressureLatest()
.observeOn(Schedulers.computation())
.subscribe(event -> compute(event.x, event.y), Throwable::printStackTrace);
```

如果在这里使用 onBackpressureDrop，会导致最后的点击被丢弃，用户可能会奇怪为什么业务逻辑没有执行。

C.2.3 创建支持回压的数据源

在处理回压相关的问题时，创建支持回压的数据源相对容易，因为库已经在 Flowable 上提供了处理回压的静态方法。我们可以将工厂方法分为两类：cold 类型的“生成器”（它能够基于下游的需求返回或生成元素）和 hot 类型的“推送器”（它通常会将非反应式和 / 或不支持回压的数据源桥接起来，在它们之上添加一个处理回压的层）。

1. just

最简单的能够感知回压的源可以通过 just 创建，如下所示。

```
Flowable.just(1).subscribe(new DisposableSubscriber<Integer>() {
    @Override
    public void onStart() {
        request(0);
    }

    @Override
    public void onNext(Integer v) {
        System.out.println(v);
    }

    //为了保持简洁，略去了剩余内容
}
```

因为我们在 onStart 中显式不请求任何值，这里不会打印出任何内容。如果我们想要为常量值启动一个序列，那么 just 是非常有用的。

但是，just 经常被误认为动态计算 Subscriber 消费的一种方式。

```
int counter;

int computeValue() {
    return ++counter;
}

Flowable<Integer> o = Flowable.just(computeValue());

o.subscribe(System.out:println);
o.subscribe(System.out:println);
```

可能会出乎一些人的预料，这里会打印两次 1，而不是分别打印出 1 和 2。如果我们重写一下调用方式，其运行方式就会更加明显。

```
int temp = computeValue();

Flowable<Integer> o = Flowable.just(temp);
```

computeValue 是主例程调用的一部分，而不在对订阅者的订阅操作的响应中。

2. fromCallable

我们真正需要的是 fromCallable。

```
Flowable<Integer> o = Flowable.fromCallable(() -> computeValue());
```

在这里，只有每个订阅者都订阅的时候，computeValue 才会执行，分别打印出 1 和 2。显然，fromCallable 能够很好地支持回压，除非请求值，否则它不会发布计算后的值。但是需要注意，无论如何计算都会发生。如果计算本身应该延迟到下游真正请求时才执行，我们可以组合使用 just 和 map。

```
Flowable.just("This doesn't matter").map(ignored -> computeValue())...
```

在真正请求之前，just 不会发布它的常量值；只有在请求的时候，它才会映射为 computeValue 的结果，此时依然会为每个订阅者分别调用。

3. fromArray

如果数据已经能够以对象数组、对象列表或 Iterable 源的形式获取，那么对应的 from 重载形式将会处理回压和这类源的发布。

```
Flowable.fromArray(1, 2, 3, 4, 5).subscribe(System.out::println);
```

为方便起见（同时避免泛型数组创建的告警），just 有 2 到 10 个参数的重载形式，它们内部都将功能委托给了 from。

fromIterable 也创造了一个很有意思的机会。很多值的生成都能以状态机的形式表述。每个元素请求都会触发状态的转换，并计算返回值。

将这样的状态机编写成 Iterable 比较复杂（不过依旧比编写成一个 Flowable 来消费它简单一些），而且与 C# 不同，Java 并没有从编译器层面对构建这种状态机提供支持，即经典风格的代码（使用 yield return 和 yield return）。有些库能够提供一些帮助，比如 Google Guava 的 AbstractIterable 和 IxJava 的 Ix.generate() 与 Ix.forloop()。这些内容本身就值得写一个完整的系列，以下是非常基础的 Iterable 源，可以无限重复地生成某个常量。

```
Iterable<Integer> iterable = () -> new Iterator<Integer>() {
    @Override
    public boolean hasNext() {
        return true;
    }

    @Override
    public Integer next() {
        return 1;
    }
};

Flowable.fromIterable(iterable).take(5).subscribe(System.out::println);
```

如果我们通过经典的 for 循环使用 iterator，将导致无限循环。但是我们在它的外部构建了一个 Flowable，可以表达只消费前 5 个流的意愿，然后停止请求任何内容。这就是在 Flowable 内部延迟执行和计算的真正威力。

4. generate()

有时候，要转换到反应式领域的数据源本身是同步（阻塞式）和拉取类型的，换言之，我们需要调用 get 或 read 方法才能获取下一块数据。当然可以将其转换为一个 Iterable，但是这种源与资源关联的时候，如果下游在序列完成之前就取消订阅，可能会造成资源泄漏。

为了处理这种问题，RxJava 提供了 generate 工厂方法家族。

```
Flowable<Integer> o = Flowable.generate(
    () -> new FileInputStream("data.bin"),
```

```
        (inputstream, output) -> {
            try {
                int abyte = inputstream.read();
                if (abyte < 0) {
                    output.onComplete();
                } else {
                    output.onNext(abyte);
                }
            } catch (IOException ex) {
                output.onError(ex);
            }
            return inputstream;
        },
        inputstream -> {
            try {
                inputstream.close();
            } catch (IOException ex) {
                RxJavaPlugins.onError(ex);
            }
        }
    );
```

一般情况下，generate 会使用 3 个回调。

第一个回调能够创建每个订阅者的专有状态，比如本例中的 FileInputStream，每个订阅者均会独立地打开该文件。

第二个回调接收这个状态对象，并且提供一个输出 Observer，可以调用它的 onXXX 方法来发布值。这个回调的执行次数与下游请求次数相同。在每次调用的时候，它最多调用一次 onNext，然后紧跟着 onError 或 onComplete。在本例中，如果读取字节为负数，我们会调用 onComplete()，表明文件的结束；如果读取抛出 IOException，会调用 onError。

在下游取消订阅（关闭输入流）或者在前面的回调调用终端方法时，最后一个回调被调用。它可以对资源进行释放，因为并非所有的源都需要这些特性，所以静态的 Flowable.generate 方法允许我们不考虑这些创建实例。

令人遗憾的是，很多方法调用会跨越 JVM，其他库会抛出检查型异常。需要将其包装到 try-catch 中，因为这个类使用的函数式接口不可以抛出检查型异常。

当然，我们也可以使用它模拟其他典型的源，比如无界的范围。

```
    Flowable.generate(
        () -> 0,
        (current, output) -> {
            output.onNext(current);
            return current + 1;
        },
        e -> { }
    );
```

在上面的代码中，current 从 0 开始，lambda 表达式下次被调用的时候，current 参数存放的值为 1。（备注：1.*x* 类 SyncOnSubscribe 和 AsyncOnSubscribe 不再可用。）

5. create(emitter)

有时候，需要使用 Flowable 包装的源已经是 hot 类型（比如鼠标的移动），或者虽然是 cold 类型，但是它的 API 不支持回压（比如异步的网络调用）。

为了处理这种场景，RxJava 最近的版本引入了 create(emitter) 工厂方法。它会接收两个参数。

- 一个回调，针对每个传入的订阅者都执行带有 Emitter<T> 接口的实例。
- 一个 BackpressureStrategy 枚举值，它会要求开发人员指定一个要使用的回压行为。它有常用的模式，类似于 onBackpressureXXX，但是它会发送 MissingBackpressureException 或者简单地忽略内部的溢出。

需要注意，它不支持为这些回压模式指定额外的参数。如果我们需要自定义行为，那么可以使用 NONE 回压模式，并在最终的 Flowable 上使用相关的 onBackpressureXXX。

它的第一个典型用例：想要与基于推送的源进行交互时，比如 GUI 事件。这些 API 会提供某种形式的 addListener/removeListener 调用以供我们使用。

```
Flowable.create(emitter -> {
    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);

    emitter.setCancellation(() ->
        button.removeListener(al));

}, BackpressureStrategy.BUFFER);
```

Emitter 使用起来非常简单。可以调用 onNext、onError 和 onComplete，这个操作符会自己处理回压和取消订阅管理。除此之外，如果包装的 API 支持取消（比如样例中的监听器移除），那么我们可以使用 setCancellation（或 Subscription 类型资源的 setSubscription）来注册取消的回调。这个回调会在下游取消订阅或给定的 Emitter 实例调用 onError/onComplete 的时候触发。

这些方法只允许 emitter 与一个资源关联，如果设置一个新的回调，原有的回调会自动取消订阅。如果必须要处理多个资源，那么可以创建一个 CompositeSubscription，与 emitter 关联，将更多的资源添加到 CompositeSubscription 本身。

```
Flowable.create(emitter -> {
    CompositeSubscription cs = new CompositeSubscription();

    Worker worker = Schedulers.computation().createWorker();

    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);
```

```
        cs.add(worker);
        cs.add(Subscriptions.create(() ->
            button.removeActionListener(al));

        emitter.setSubscription(cs);

    }, BackpressureMode.BUFFER);
```

第二个用例通常涉及一些异步的、基于回调的 API，需要将其转换为 Flowable。

```
    Flowable.create(emitter -> {

        someAPI.remoteCall(new Callback<Data>() {
            @Override
            public void onSuccess(Data data) {
                emitter.onNext(data);
                emitter.onComplete();
            }

            @Override
            public void onFailure(Exception error) {
                emitter.onError(error);
            }
        });

    }, BackpressureMode.LATEST);
```

在这种情况下，代理按照相同的方式运行。但是，这些基于回调风格的 API 通常不支持取消；不过，如果它们能够支持取消，我们就可以按照前面样例的方式设置取消功能（当然可能会涉及更复杂的方式）。注意，这里使用了 LATEST 回压模式。如果我们明确知道只有一个值，那么就不需要使用 BUFFER 策略，因为它默认分配 128 个元素（随需要而增长）的缓冲区，该缓冲区永远不会被充分利用。

关于作者

托马什·努尔凯维茨（Tomasz Nurkiewicz）是 Allegro 的一名软件工程师。在过去的十年里，他一直在从事 Java 编程，并且热爱后端开发技术。他热爱 JVM 语言和开源技术。他还经常为 DZone 网站撰写博客，并在世界各地的 Java 会议上发表演讲。可以通过 Twitter 账号 @tnurkiewicz 和博客（https://www.nurkiewicz.com/）联系他。

本·克里斯滕森（Ben Christensen）是一名专注于弹性、扩展性和分布式系统的软件工程师。他创建了满足这些需求的开源项目，包括 Hystrix 和 RxJava。

关于封面

本书封面上的动物是巢鼬，又名南美狼獾。

巢鼬体长可达 60 厘米左右，体重 0.9~2.7 千克。巢鼬有两个现存物种——大巢鼬和小巢鼬，二者的主要区别在于体型的大小。巢鼬的体色很像臭鼬，白色条纹从前额一直延伸到脖子后面，但它们比臭鼬更强壮、脖子更宽、腿更短、尾巴更小。

巢鼬通常生活在半开放的灌木丛和低海拔的林地或森林中。它们在倒下的树或岩石裂缝中筑巢，主要以水果和小动物为食。

O'Reilly 图书封面上的许多动物都属于濒危物种，它们都是这个世界不可或缺的一部分。如果想要了解如何为这些动物提供帮助，请访问 http://animals.oreilly.com。

封面图片来自一幅匈牙利插图。

O'REILLY®

RxJava反应式编程

如今，移动App驱动着我们的生活，程序的异步性和响应式至关重要。反应式编程技术能够帮助我们编写易于扩展、性能良好且可靠性强的代码。在这本注重实战的图书中，Java开发人员首先将会学习如何以反应式的方式看待问题，然后再借助这一令人兴奋的编程范式的优秀特性构建程序。

本书包含了一些使用RxJava的具体样例，用来解决Android设备和服务器端的实际性能问题。你将会学到RxJava如何借助并行和并发解决当前的问题。本书还特别收录了2.0版本的基本情况。

- 编写对多个异步源输入进行响应的程序，避免陷入“回调地狱”
- 理解如何以反应式的方式解决问题
- 处理Observable生产数据太快的问题
- 探索调试和测试反应式程序的策略
- 在程序中高效利用并行和并发
- 学习如何迁移至RxJava 2.0版本

托马什·努尔凯维茨（Tomasz Nurkiewicz），软件工程师，热爱JVM语言和开源技术，经常为DZone网站撰写博客，并在世界各地的Java会议上发表演讲。

本·克里斯滕森（Ben Christensen），软件工程师，曾在苹果、Netflix和Facebook公司工作，专注于弹性、扩展性和分布式系统，为Hystrix和RxJava等开源项目做出了贡献。

“这本书深入探讨了RxJava的理念和用法，以及反应式编程的通用知识。两位作者在实现和使用RxJava方面拥有丰富的经验。如果你想掌握反应式编程，那么没有比阅读这本书更好的方法了。”

——Erik Meijer
Applied Duality公司总裁兼创始人

“对于现代Android应用程序来讲，高度状态化、并发和异步实现是基本的特性，而RxJava是管理它们的好工具。这本书既是一个渐进式的学习工具，也是一份随时可翻阅的参考资料，如果没有它，掌握RxJava这个库可能会非常困难。”

——Jake Wharton
Square公司软件工程师

REACTIVE PROGRAMMING

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议　计算机 / 程序设计 / RxJava

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-52400-3
9 787115 524003 >
ISBN 978-7-115-52400-3
定价：99.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks